# 2021.11.06. 교육

나정휘
https://justiceHui.github.io/

# 목차

# 계산 기하

# 계산 기하

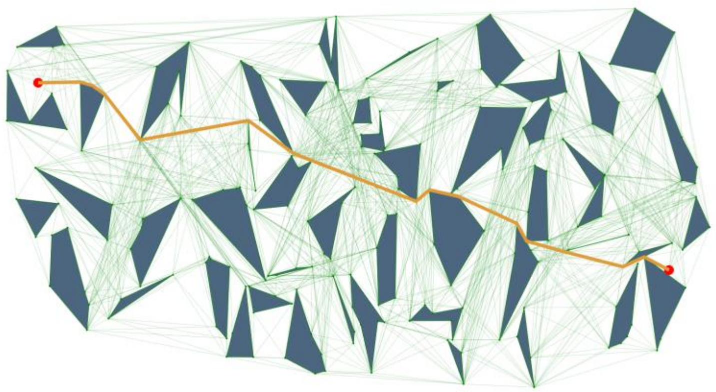
- 기하학적 도형을 대상으로 하는 문제를 알고리즘적 방법으로 해결하는 분야
  - 점
  - 직선, 반직선, 선분, 곡선
  - 다각형, 원
  - 다면체, 구

# 계산 기하

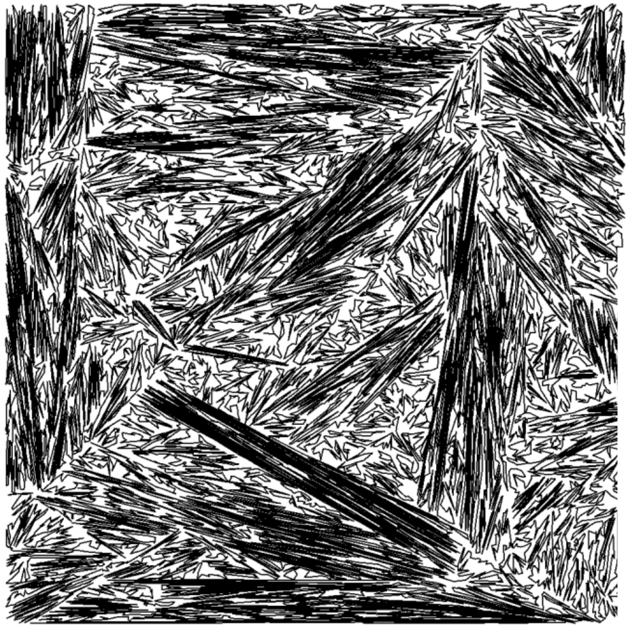
- 기하학적 도형을 대상으로 하는 문제를 알고리즘적 방법으로 해결하는 분야
  - 점
  - 직선, 반직선, 선분, 곡선
  - 다각형, 원
  - 다면체, 구

# 계산 기하

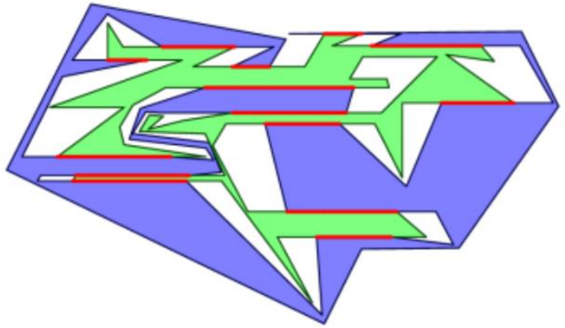
- 구현하기 귀찮음
- 수많은 예외 처리
- 쉬운 것만 다룰 예정



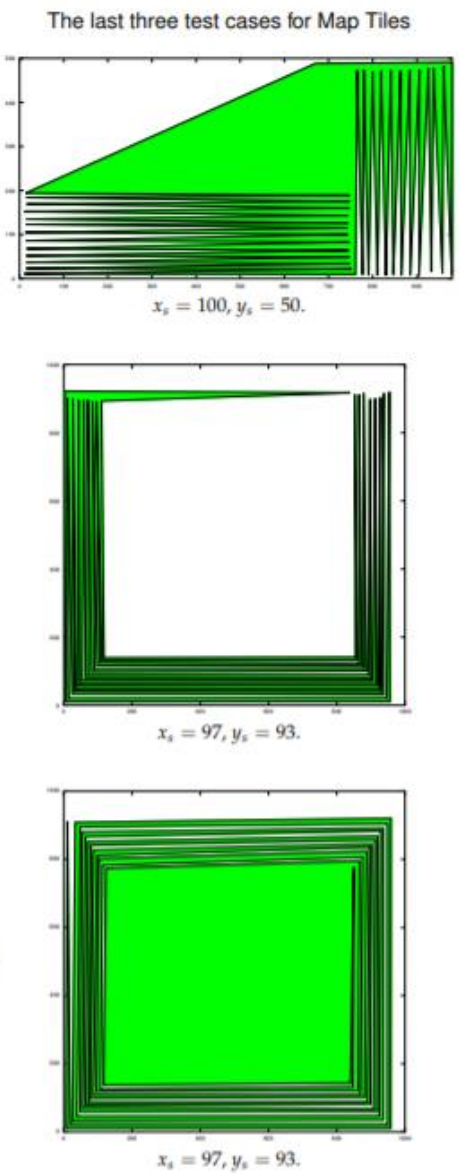
visibility graph



놀랍게도 다각형



두 "단순" 다각형의 접선




ㅎㅎ;;

# 2차원 물체의 표현

# 2차원 물체의 표현

- 3차원 부터는 웬만하면 올림피아드에 안 나옴
  - 왜?
  - 3차원은 많이 어렵고 귀찮다.
  - 2차원으로 충분히 어려운 문제를 많이 낼 수 있다.

- 0~2차원에서 사용하는 물체들
  - 0차원 : 점
  - 1차원 : 직선, 선분, ~~곡선~~
  - 2차원 : 다각형, ~~원, 곡면~~

# 2차원 물체의 표현

- 점
    - 좌표 평면 위에 있는 점
    - (x, y)
    - std::pair

```cpp
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;

using Point = pair<int, int>;

int main(){
    Point pt(1, 2);
    cout << pt.x << " " << pt.y << endl;
}
```

# 2차원 물체의 표현

- 두 점 (x1, y1), (x2, y2)를 지나는 직선

- y = ax + b
  - 일차 함수로 표현
  - 기울기 a = (y2 – y1) / (x1 – x2)
  - 절편 b = y1 – mx1
  - 장점 : 교점을 구하기 쉬움, 점과 동일하게 취급할 수 있음
  - 단점 : 기울기가 무한대일 때 예외 발생, 선분을 표현하기 힘듬

  - ax + by + c = 0으로 표현하는 경우도 존재
  - 기울기 무한대를 표현할 수 있지만 수식이 더러워짐

# 2차원 물체의 표현

- 두 점 (x1, y1), (x2, y2)를 지나는 직선
- pair<Point, Point>
  - 두 점을 지나는 직선이니까 두 점을 저장한다는 아이디어
  - 장점 : 선분을 표현할 때 편함
  - 단점 : 직선을 표현할 때 불편함

# 2차원 물체의 표현

- 두 점 (x1, y1), (x2, y2)를 지나는 직선
- x(t), y(t)
  - 매개변수로 표현
  - x(t) = x1 + (x2 – x1)t
  - y(t) = y1 + (y2 – y1)t
  - -inf < t < inf면 직선을 표현
  - 0 ≤ t ≤ 1이면 선분을 표현
  - 장점 : 직선 위의 점을 깔끔하게 표현, 예외처리가 적음
  - 단점 : 익숙하지 않으면 실수하기 쉬움

# 2차원 물체의 표현

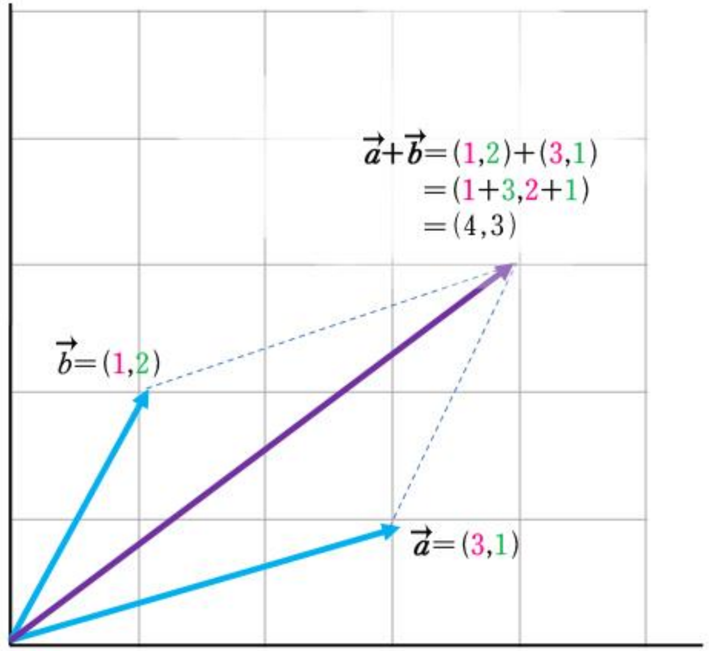
- 다각형
  - 다각형에 속한 점 p1, p2, p2, … , pn을 순서대로 배열에 저장
  - 어떤 순서대로?
  - 보통 반시계 방향으로 저장함

# 질문

CCW

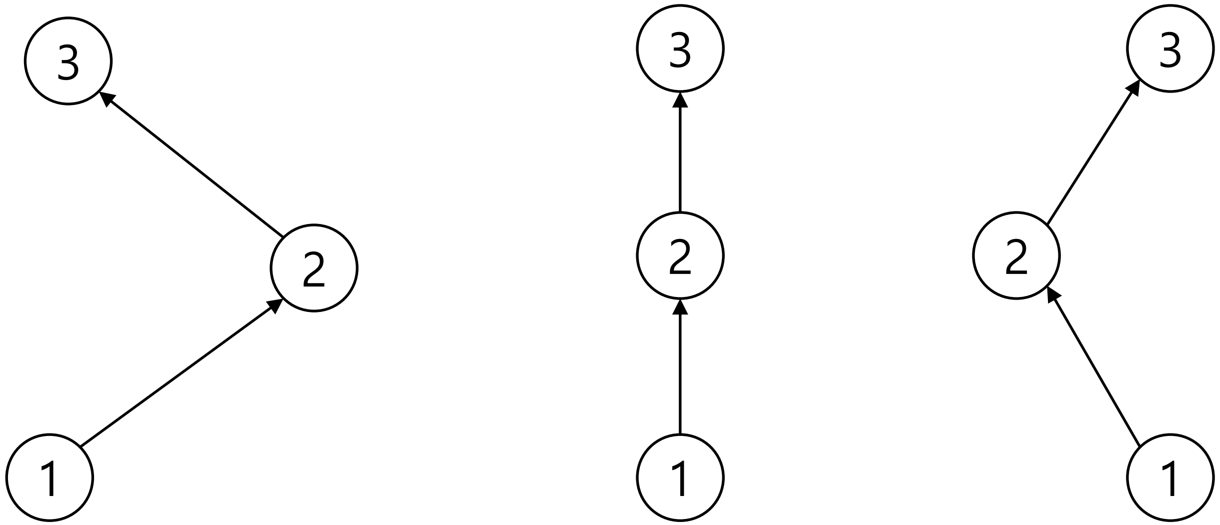
# 벡터

- 사실 점은 벡터로 표현할 수 있음 (방향, 크기)
- 벡터의 덧셈, 뺄셈
- 벡터의 내적, 외적 (?)



# CCW

- CCW (CounterClockWise)
  - 두 벡터가 시계 방향인지 반시계 방향인지 판별하는 방법
  - 시계 방향이면 음수, 반시계 방향이면 양수, 일직선이면 0 반환



# CCW

- 신발끈 공식
  - 세 점 A(x1,y1), B(x2,y2), C(x3,y3)을 꼭짓점으로 하는 삼각형의 넓이
  - ½ | (x1y2+x2y3+x3y1) – (x2y1+x3y2+x1y3) |
  - 절댓값을 붙이는 이유는?

- 세 점이 시계 방향이면 음수가 나오고, 반시계 방향이면 양수가 나옴
- 저 식을 그대로 사용하면 된다.
  - 또는 (x2-x1)(y3-y2) – (x3-x2)(y2-y1)

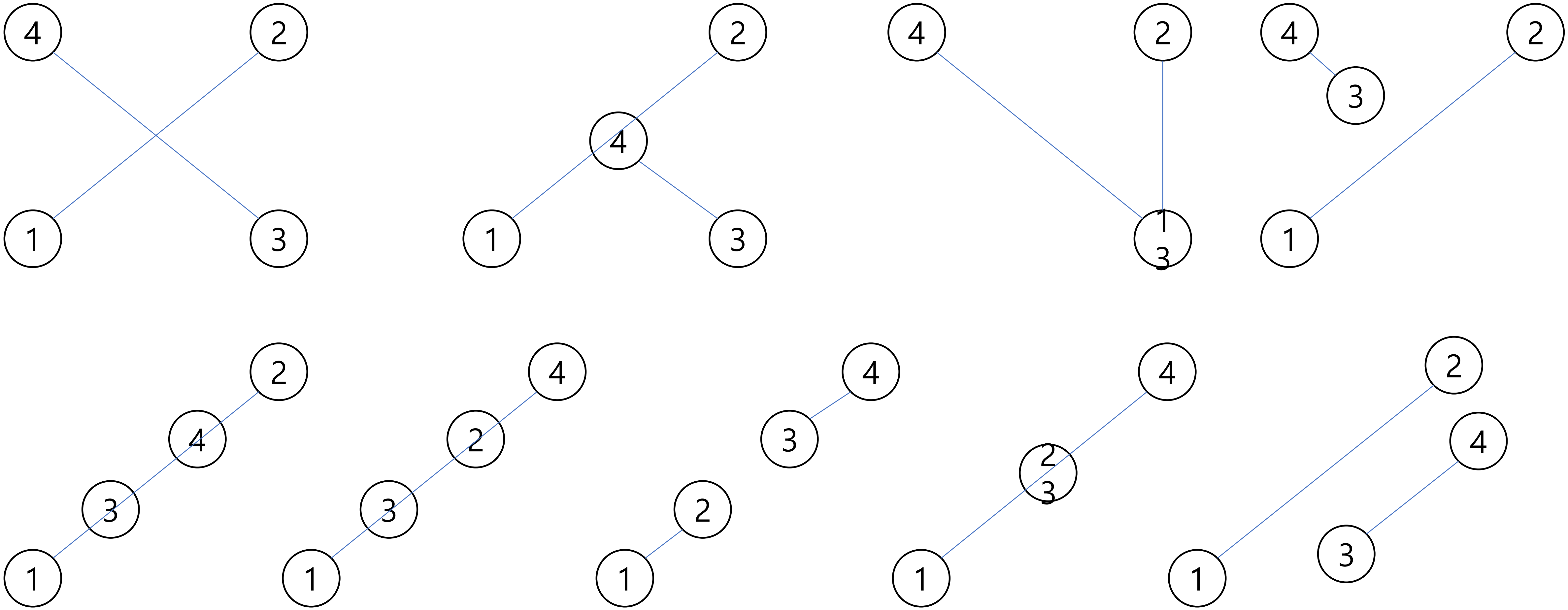
# CCW

- 오버플로우 주의

```cpp
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;

using ll = long long;
using Point = pair<ll, ll>;

int CCW(const Point &p1, const Point &p2, const Point &p3){
    ll res = p1.x*p2.y + p2.x*p3.y + p3.x*p1.y;
    res   -= p2.x*p1.y + p3.x*p2.y + p1.x*p3.y;
    return (res > 0) - (res < 0);
}
```

# 선분 교차 판별

- 두 선분이 만나는지 판별



# 선분 교차 판별

- 예외가 너무 많아서 실수하기 쉬우므로 코드를 외우는 것이 편함

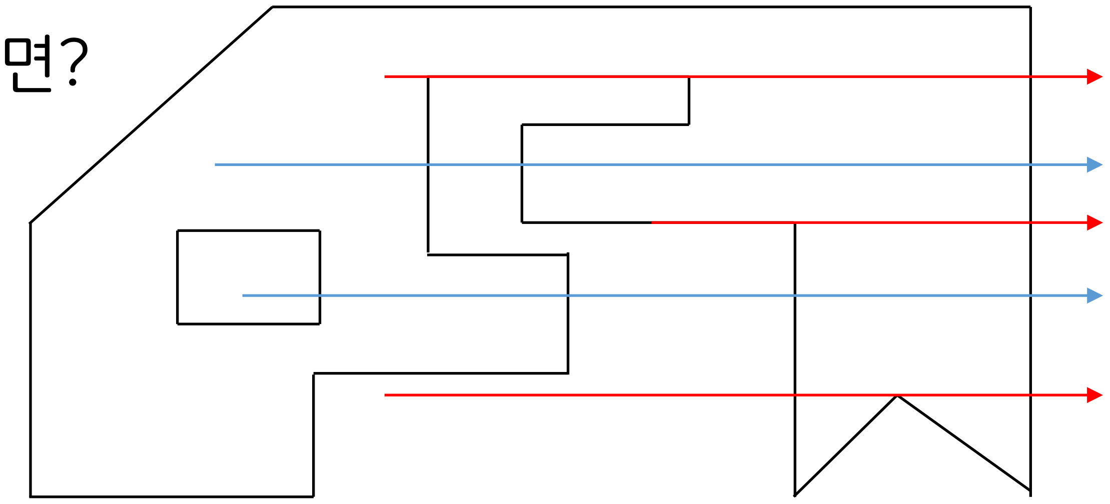
```
bool Cross(Point a, Point b, Point c, Point d){
    int ab = CCW(a, b, c) * CCW(a, b, d);
    int cd = CCW(c, d, a) * CCW(c, d, b);
    if(ab == 0 && cd == 0){
        if(a > b) swap(a, b);
        if(c > d) swap(c, d);
        return !(b < c || d < a);
    }
    return ab <= 0 && cd <= 0;
}
```

# 질문

# 다각형 내부 판별

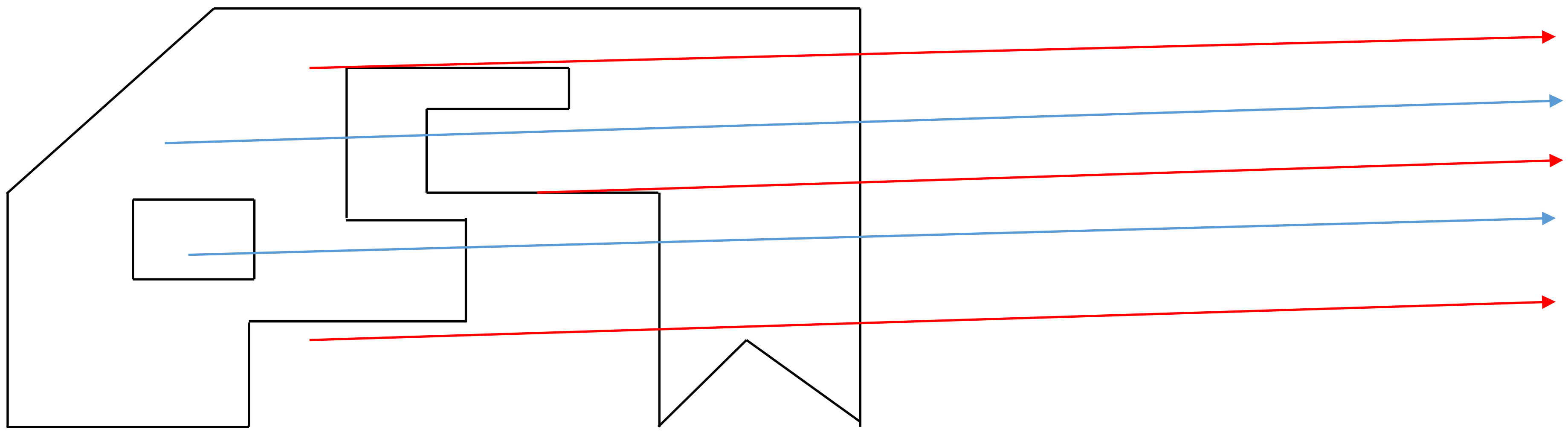
# 다각형 내부 판별

- 다각형과 어떤 점이 주어지면, 그 점이 다각형 내부에 있는지 판별
- 그 점에서 시작하는 반직선을 그린 뒤, 다각형과의 교점 개수를 세면 됨
  - 반직선은 (x, y)와 (inf, y)로 표현 가능
  - 홀수 : 내부
  - 짝수 : 외부
- 변/꼭짓점과 겹치면?



# 다각형 내부 판별

- 반직선을 (x, y)와 (inf, y+1)로 표현하면 변/꼭짓점과 겹치지 않음
  - 반직선의 기울기가 1/inf가 되어서 (x, y)를 제외한 어떠한 격자점에서도 다각형과 만나지 않음



# 다각형 내부 판별

- 반직선을 (x, y)와 (inf, y+1)로 표현하면 변/꼭짓점과 겹치지 않음
  - 반직선의 기울기가 1/inf가 되어서 (x, y)를 제외한 어떠한 격자점에서도 다각형과 만나지 않음

```
const ll MAX_COORD = 1e9;
bool Check(const vector<Point> &v, Point p1){
    int n = v.size(), cnt = 0;
    Point p2(MAX_COORD+1, p1.y+1);
    for(int i=0; i<n; i++){
        int j = i+1 == n ? 0 : i+1;
        cnt += Cross(v[i], v[j], p1, p2);
    }
    return cnt & 1;
}
```

# 질문

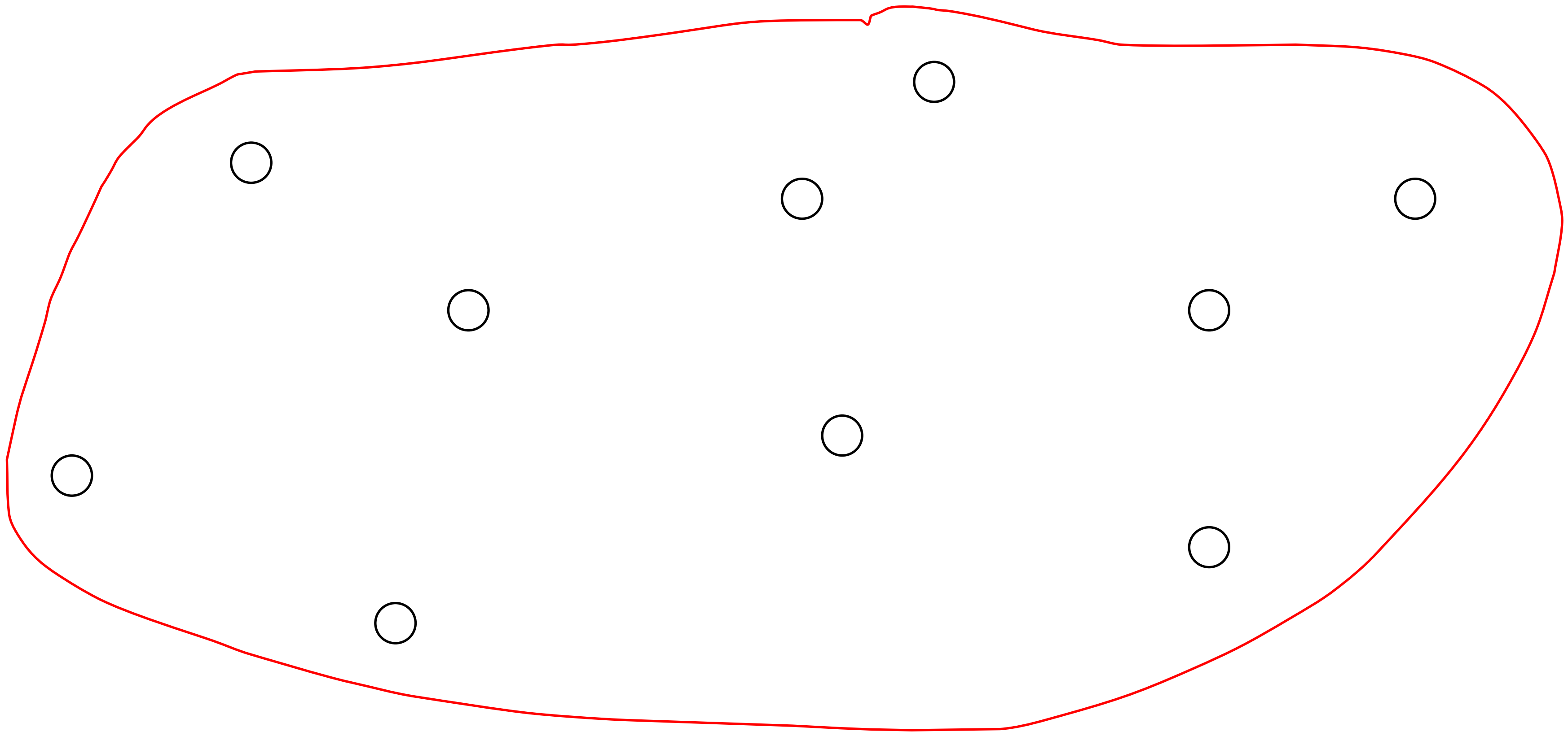
# 볼록 껍질

# 볼록 껍질

- 볼록 다각형 : 모든 내각이 180도 미만인 다각형
- 볼록 껍질 : 주어진 점들을 모두 둘러싸는 가장 작은 볼록 다각형
  - O(N log N)에 구할 수 있다!

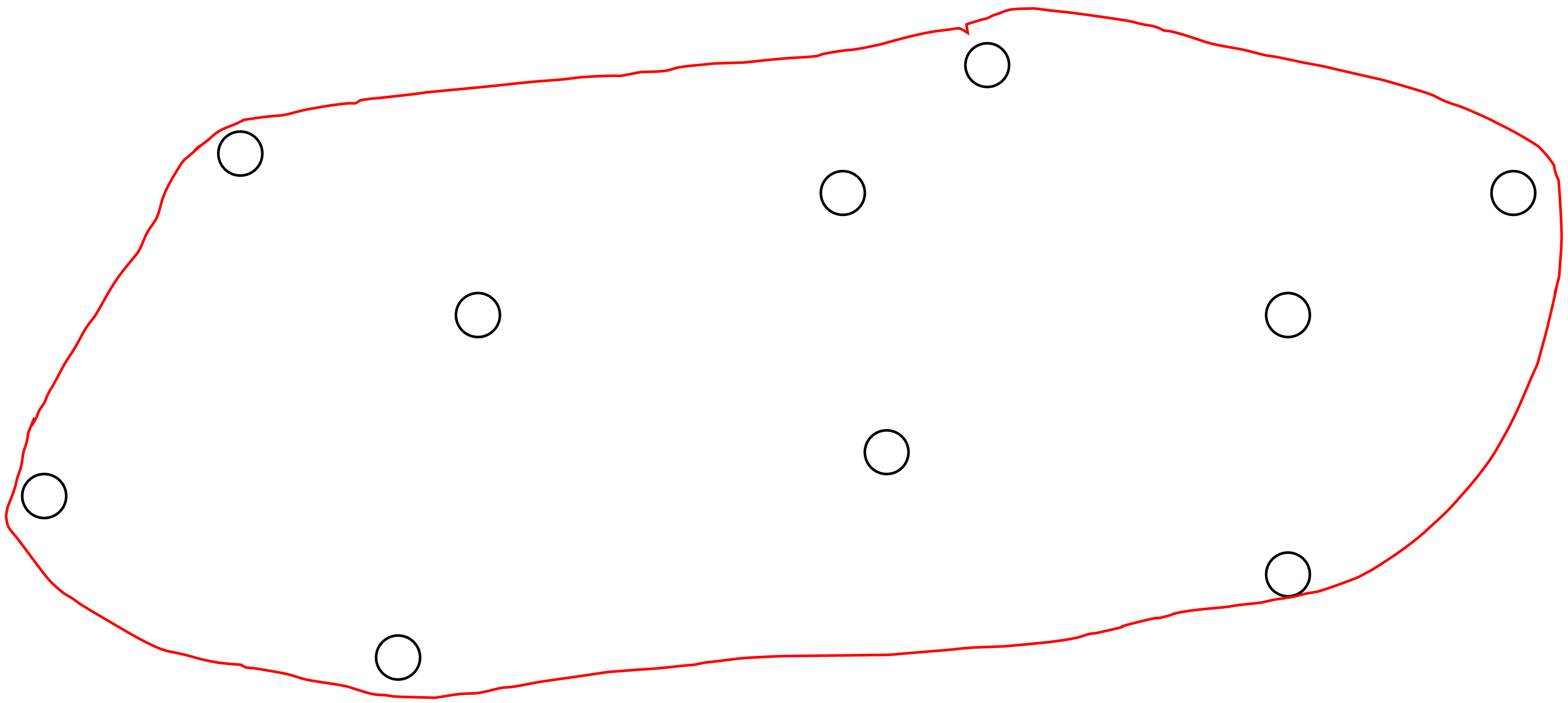
# 볼록 껍질

- 볼록 다각형 : 모든 내각이 180도 미만인 다각형
- 볼록 껍질 : 주어진 점들을 모두 둘러싸는 가장 작은 볼록 다각형



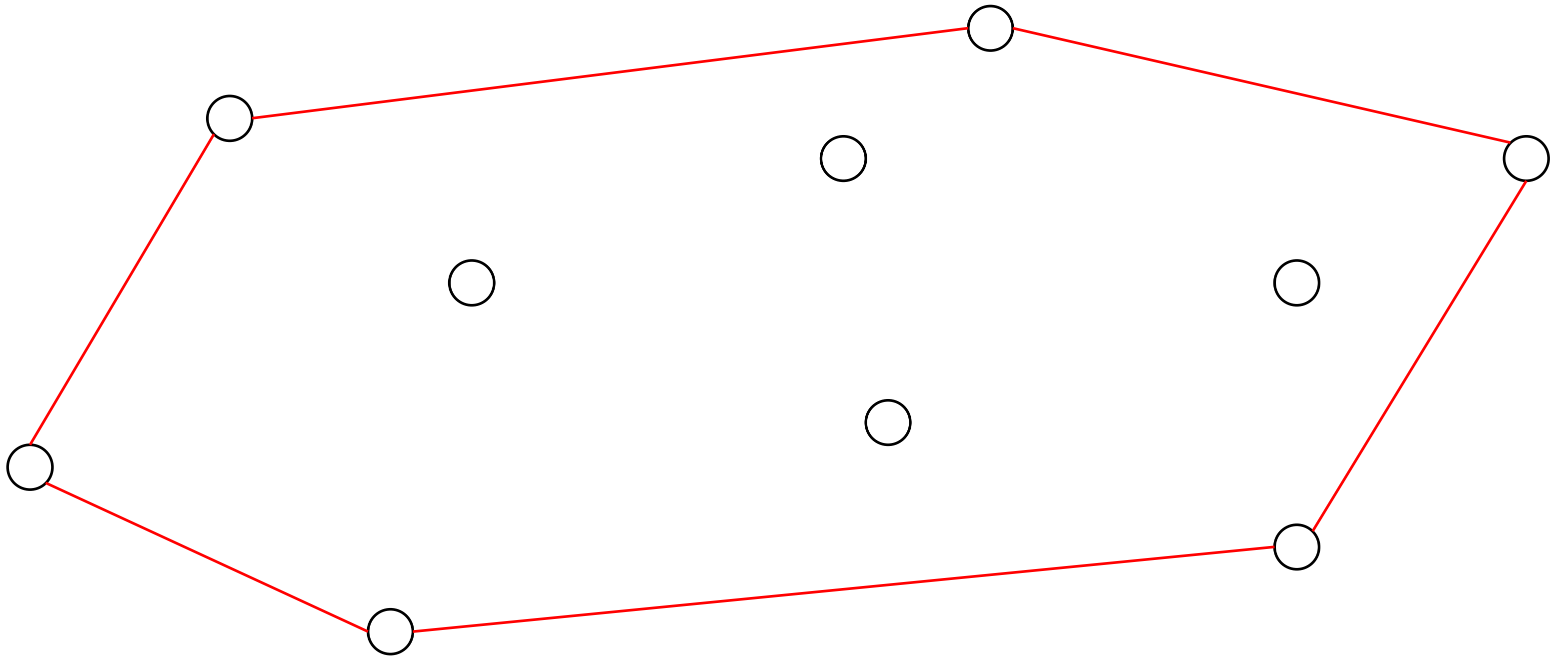
# 볼록 껍질

- 볼록 다각형 : 모든 내각이 180도 미만인 다각형
- 볼록 껍질 : 주어진 점들을 모두 둘러싸는 가장 작은 볼록 다각형



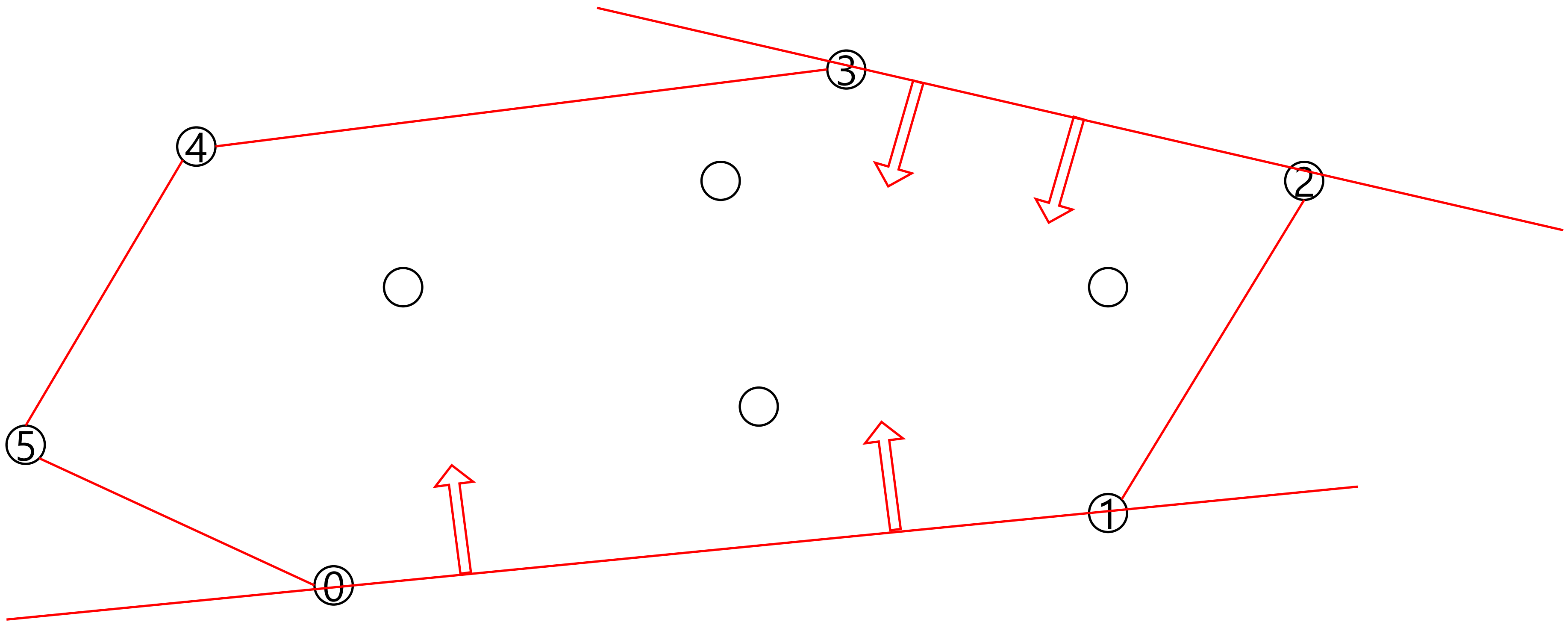
# 볼록 껍질

- 볼록 다각형 : 모든 내각이 180도 미만인 다각형
- 볼록 껍질 : 주어진 점들을 모두 둘러싸는 가장 작은 볼록 다각형



# 볼록 껍질

- 볼록 다각형 : 모든 내각이 180도 미만인 다각형
- 볼록 껍질 : 주어진 점들을 모두 둘러싸는 가장 작은 볼록 다각형

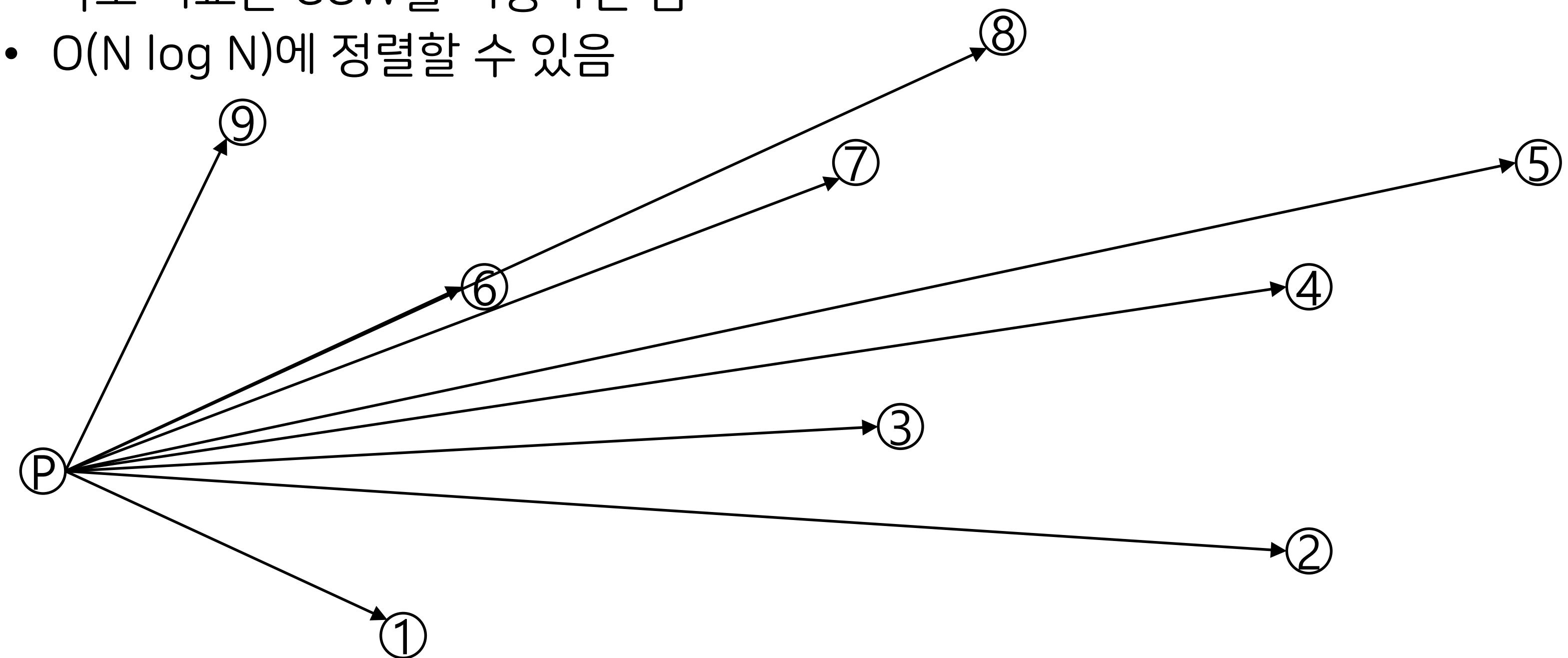


# 볼록 껍질

- O(N^3) 알고리즘
  - 모든 점 쌍을 보면서
  - 다른 모든 점들이 한 방향에 있으면 볼록 껍질의 변이 됨
  - 너무 느림

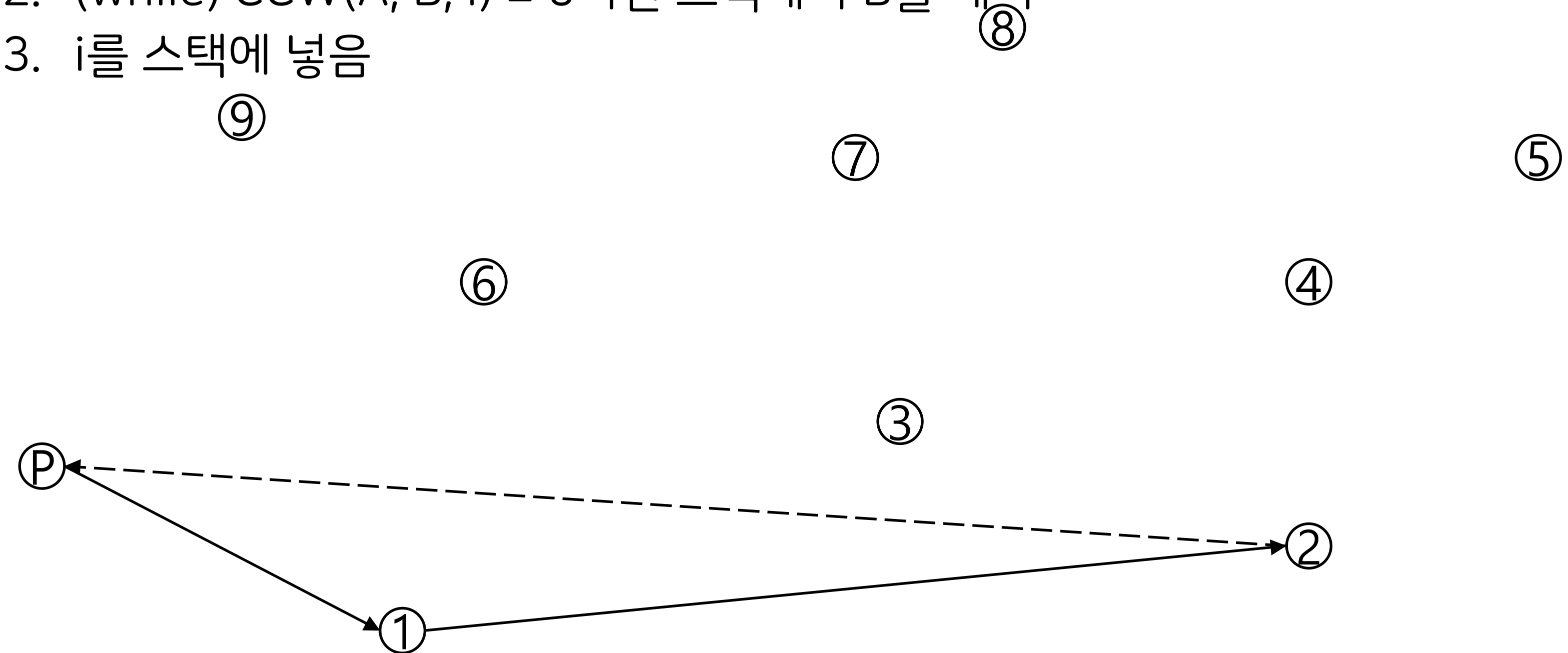
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  1. x좌표가 가장 작은 점(여러 개라면 y좌표가 가장 작은 점) P를 찾는다.
     - P는 항상 볼록 껍질의 꼭짓점, O(N)에 찾을 수 있음
  2. 다른 모든 점을 P를 기준으로 **각도 순서**대로(일직선이면 가까운 점이 먼저 오도록) 정렬한다.
     - 모든 점을 반시계 방향으로 정렬
     - 각도 비교는 CCW를 이용하면 됨
     - O(N log N)에 정렬할 수 있음



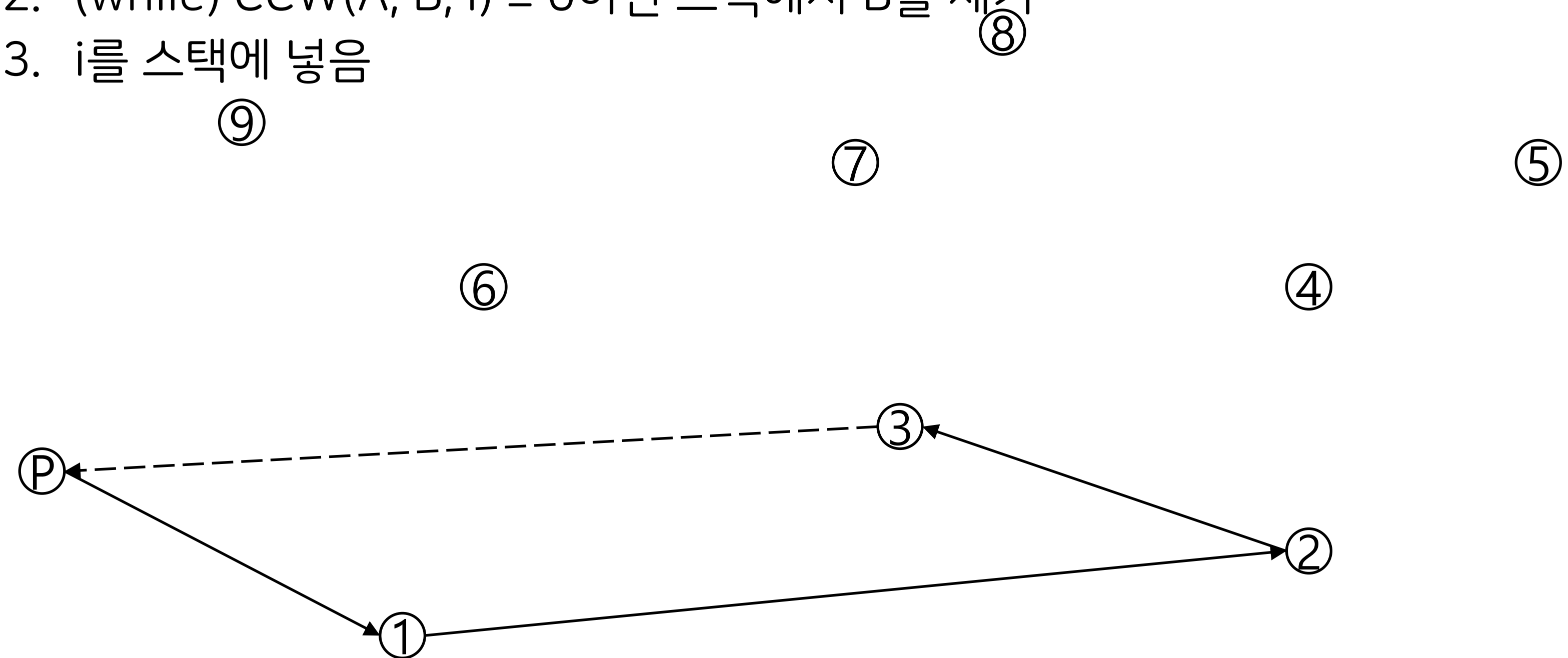
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



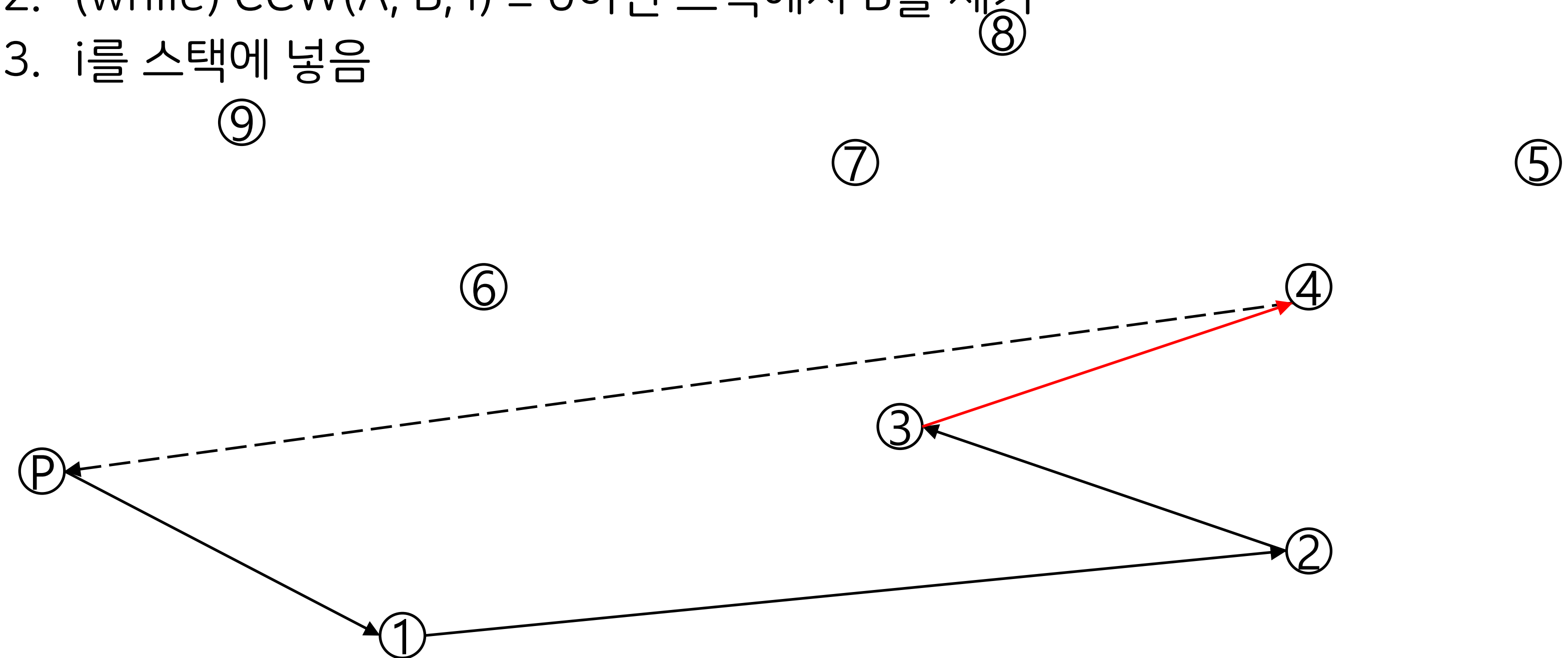
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



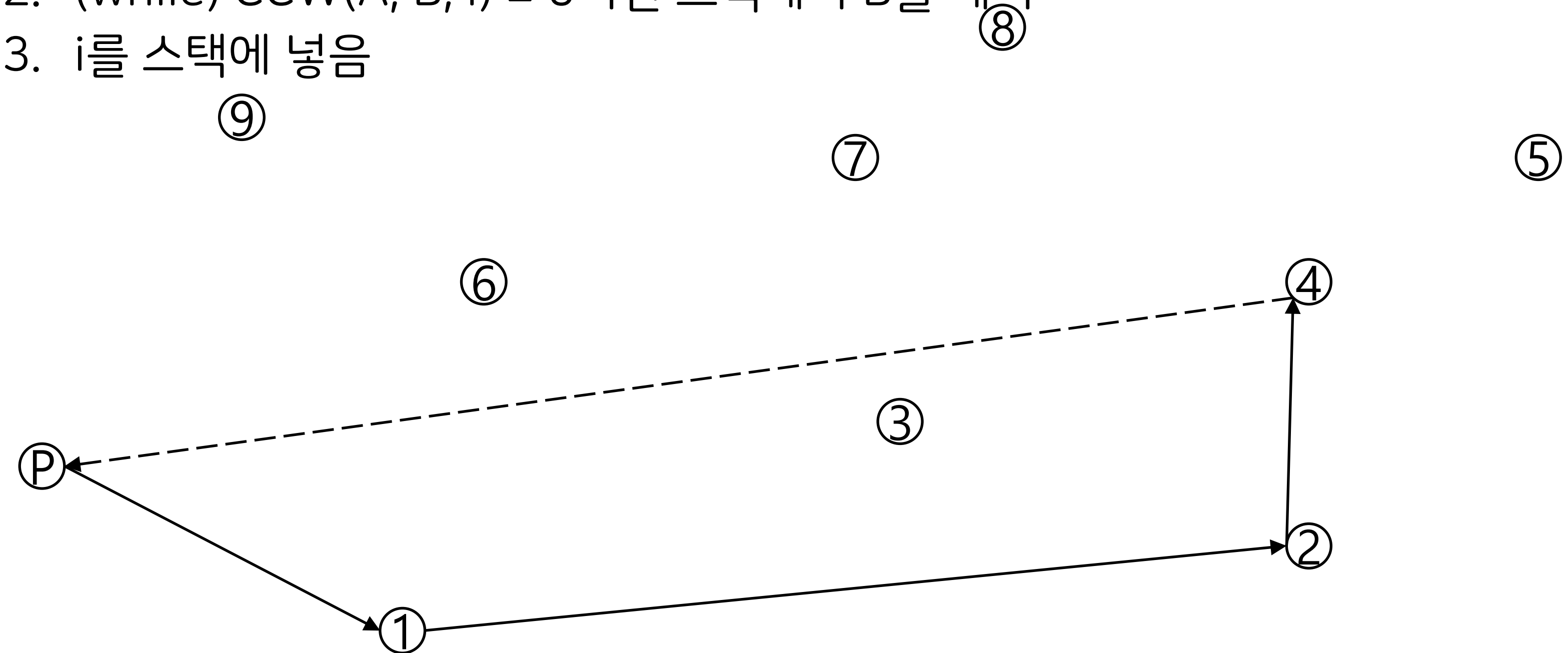
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



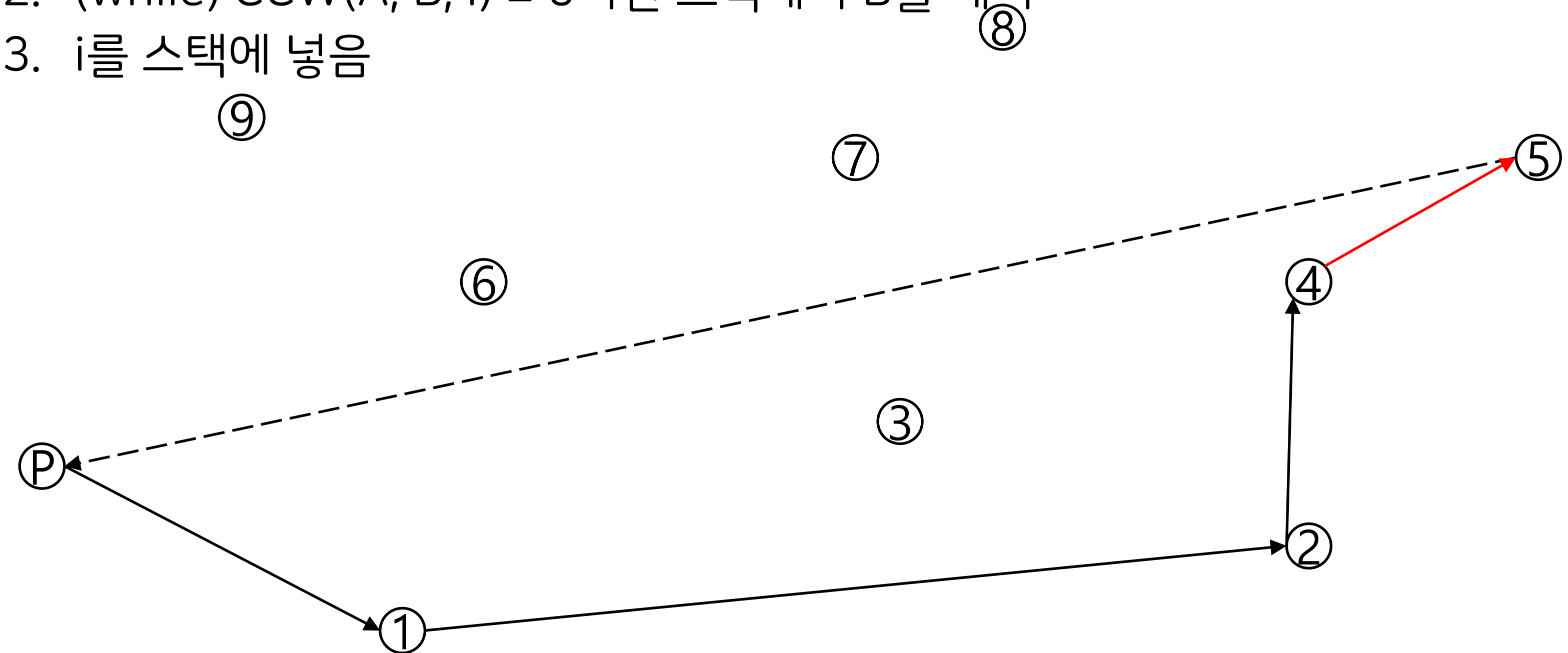
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



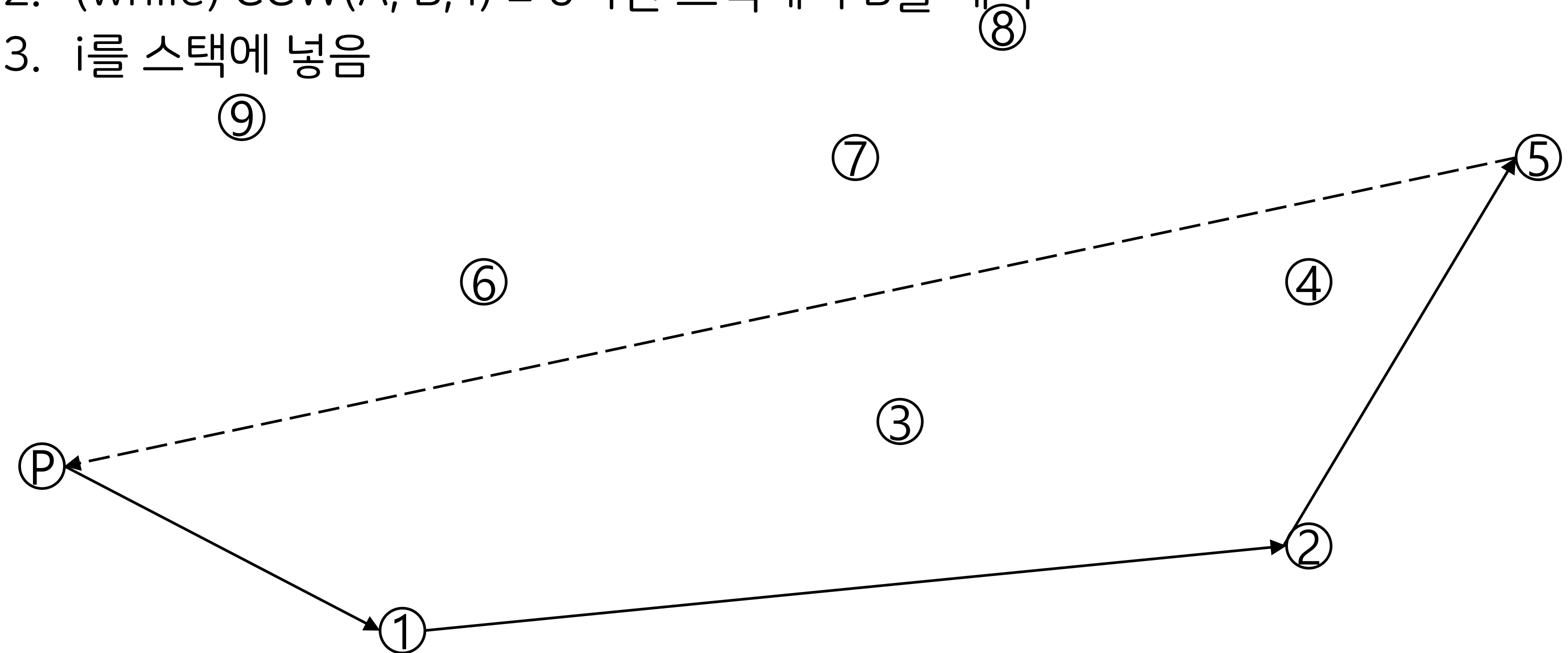
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



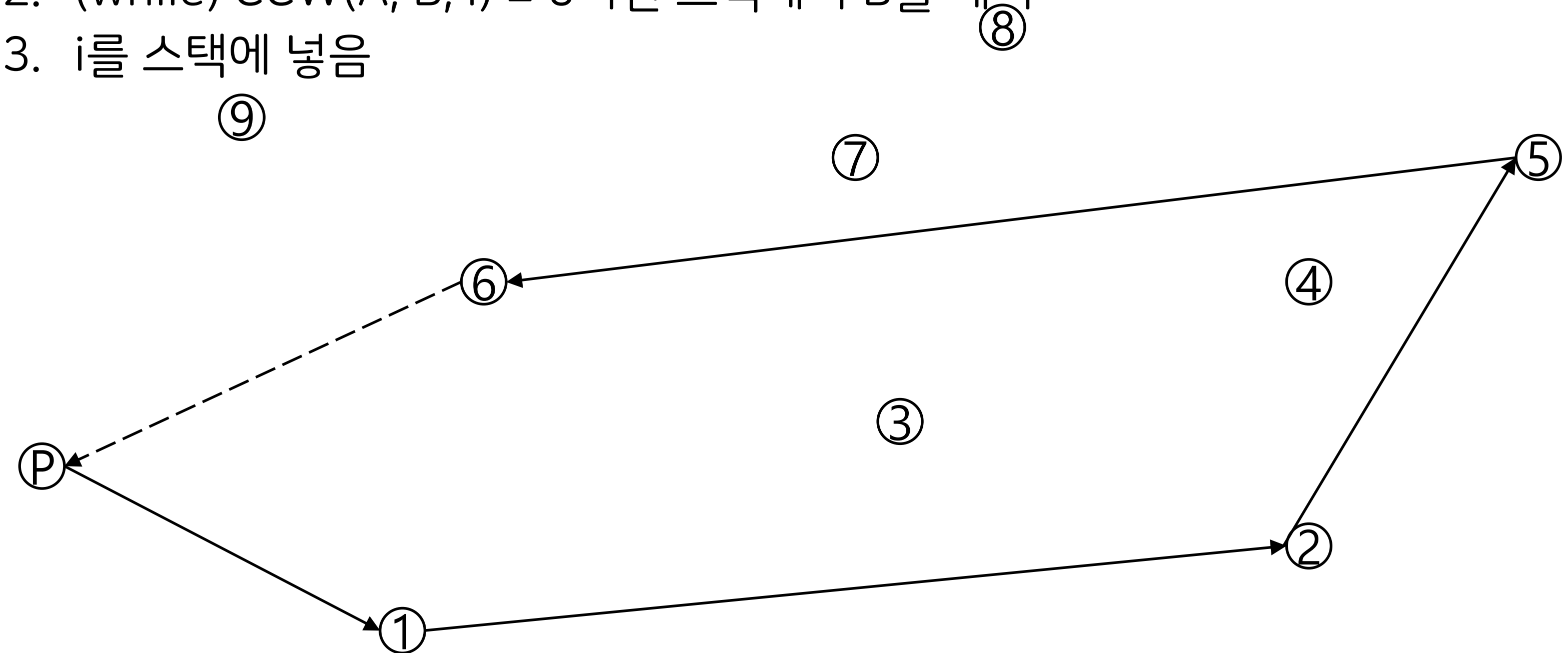
# 볼록 껍질

- Graham’s Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



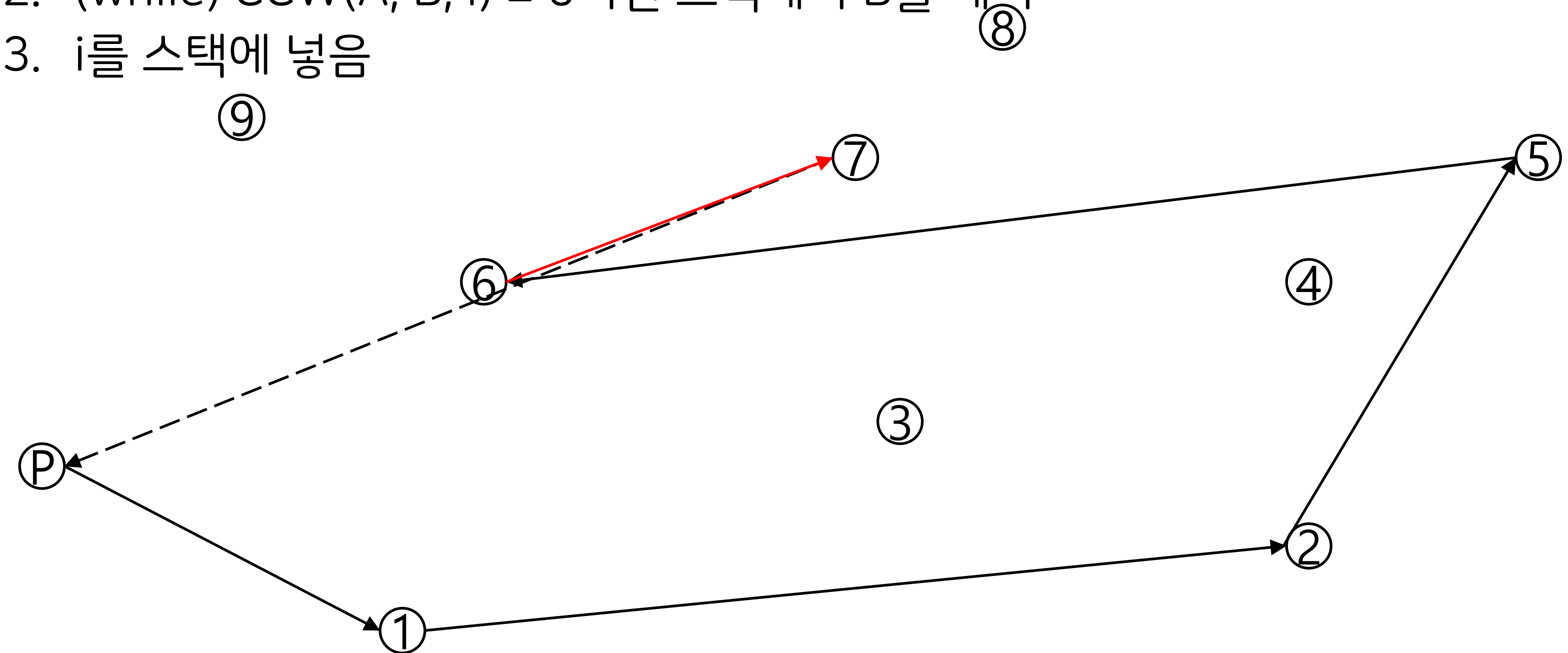
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
    - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
    - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
    - 3번 점부터 차례대로 보면서 아래 과정을 수행
        1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
        2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
        3. i를 스택에 넣음



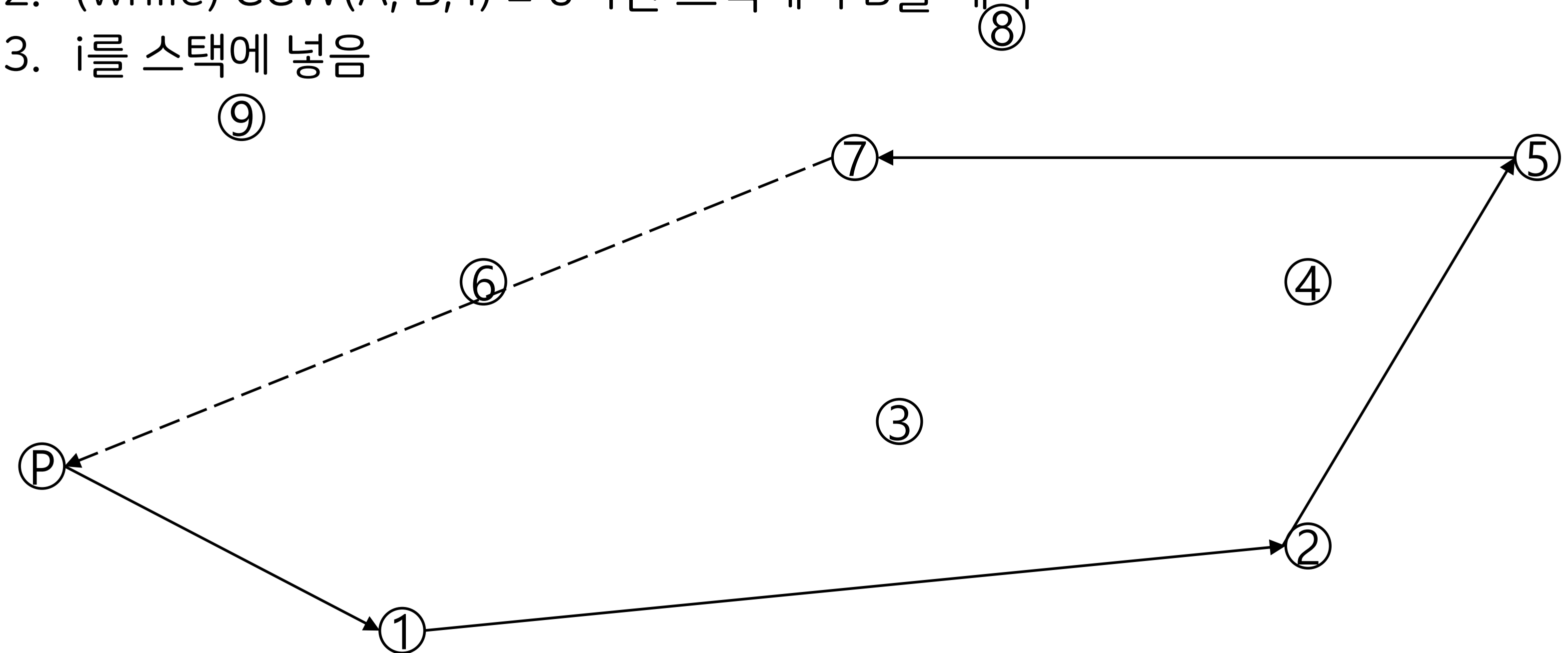
# 볼록 껍질

- Graham’s Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



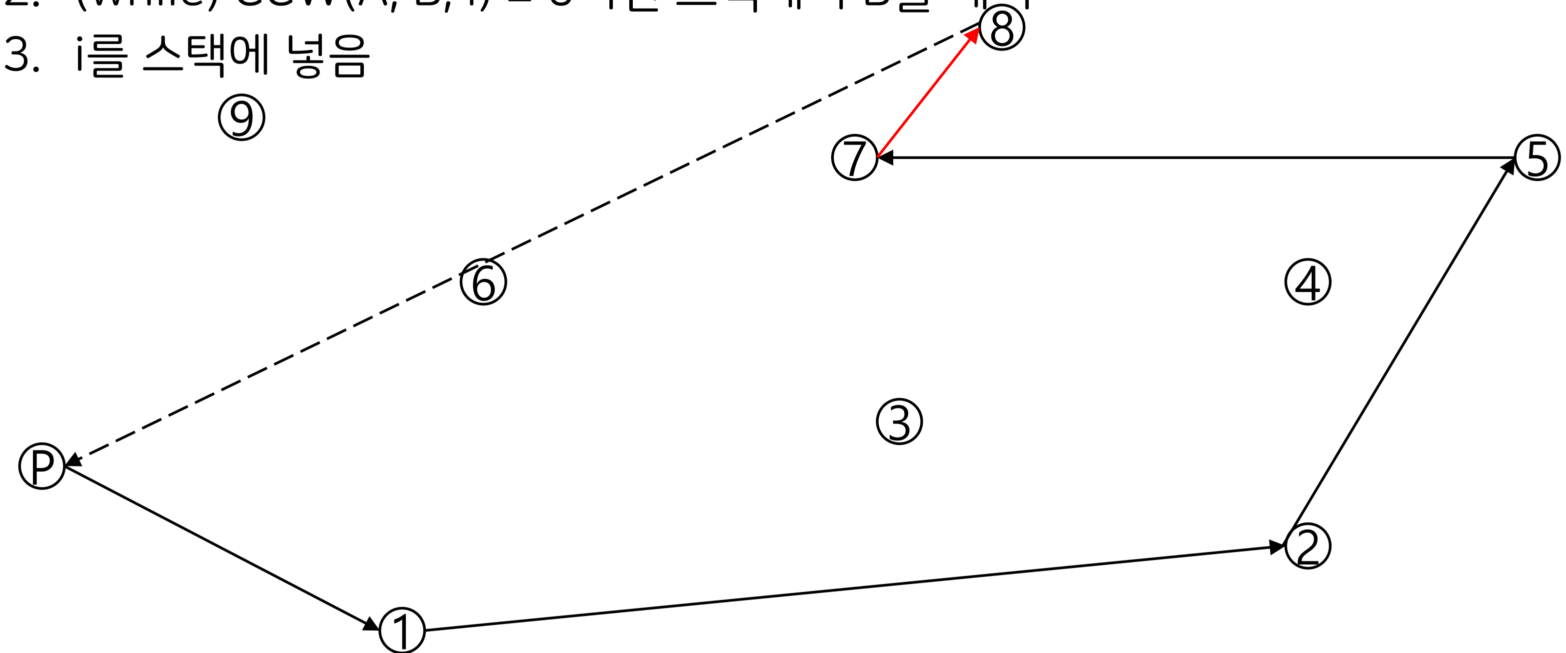
# 볼록 껍질

- Graham’s Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



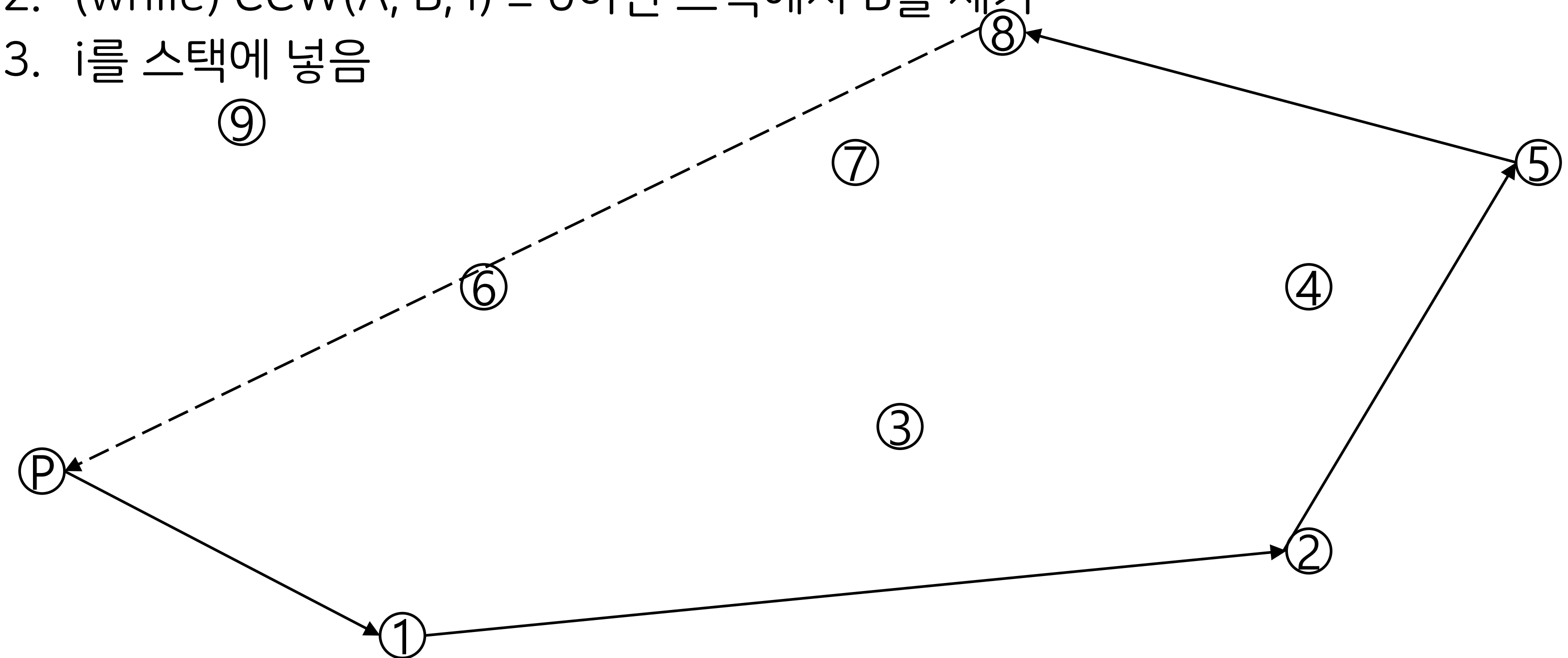
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
    - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
    - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
    - 3번 점부터 차례대로 보면서 아래 과정을 수행
        1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
        2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
        3. i를 스택에 넣음



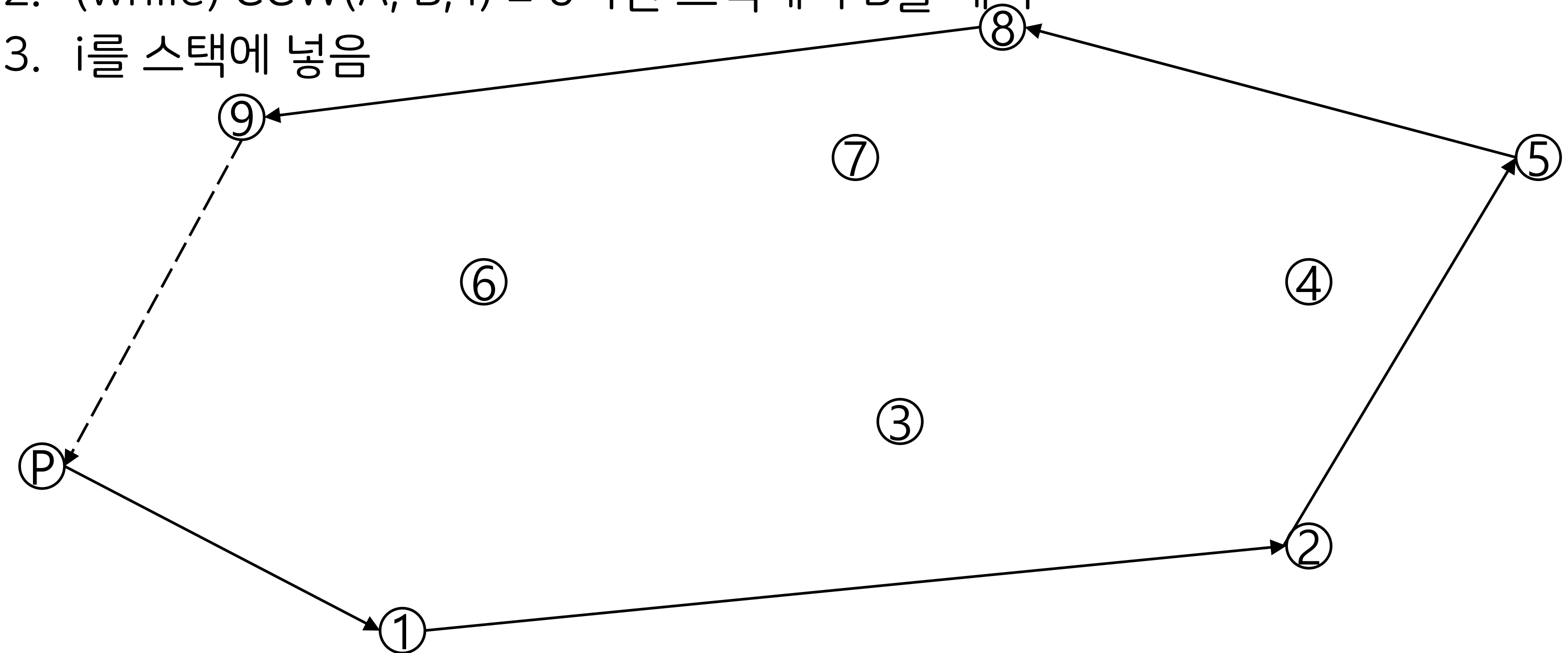
# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



# 볼록 껍질

- Graham's Scan : 0..i번째 점의 Convex Hull을 관리
  - 점을 차례대로 볼록 껍질에 넣는 방식으로 진행, 이때 볼록 껍질은 스택으로 관리
  - 처음에는 P, 1, 2번 점으로 이루어진 볼록 껍질에서 시작
  - 3번 점부터 차례대로 보면서 아래 과정을 수행
    1. 스택의 맨 뒤에 있는 점 B, 두 번째로 뒤에 있는 점 A, 현재 점 i
    2. (while) CCW(A, B, i) ≤ 0이면 스택에서 B를 제거
    3. i를 스택에 넣음



# 볼록 껍질

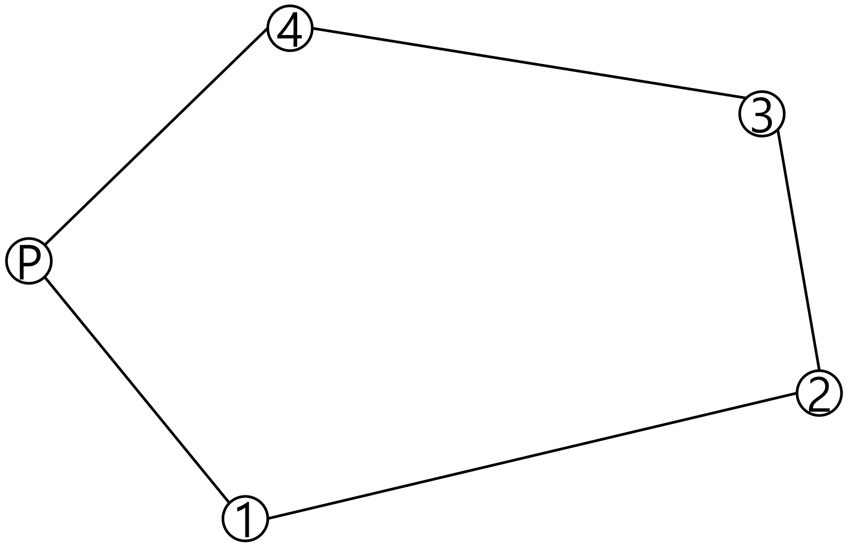
```cpp
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;
using ll = long long;
using Point = pair<ll, ll>;

int CCW(Point p1, Point p2, Point p3){
    ll res = (p2.x - p1.x) * (p3.y - p2.y) - (p3.x - p2.x) * (p2.y - p1.y);
    return (res > 0) - (res < 0);
}

ll Dist(Point p1, Point p2){
    ll dx = p2.x - p1.x, dy = p2.y - p1.y;
    return dx*dx + dy*dy;
}

vector<Point> ConvexHull(vector<Point> points){
    swap(points[0], *min_element(points.begin(), points.end()));
    sort(points.begin()+1, points.end(), [&](auto a, auto b){
        int dir = CCW(points[0], a, b);
        if(dir != 0) return dir > 0;
        return Dist(points[0], a) < Dist(points[0], b);
    });

    vector<Point> stk;
    for(auto i : points){
        while(stk.size() >= 2 && CCW(stk[stk.size()-2], stk.back(), i) <= 0) stk.pop_back();
        stk.push_back(i);
    }
    return stk;
}
```

# 질문

# 볼록 다각형 내부 판별

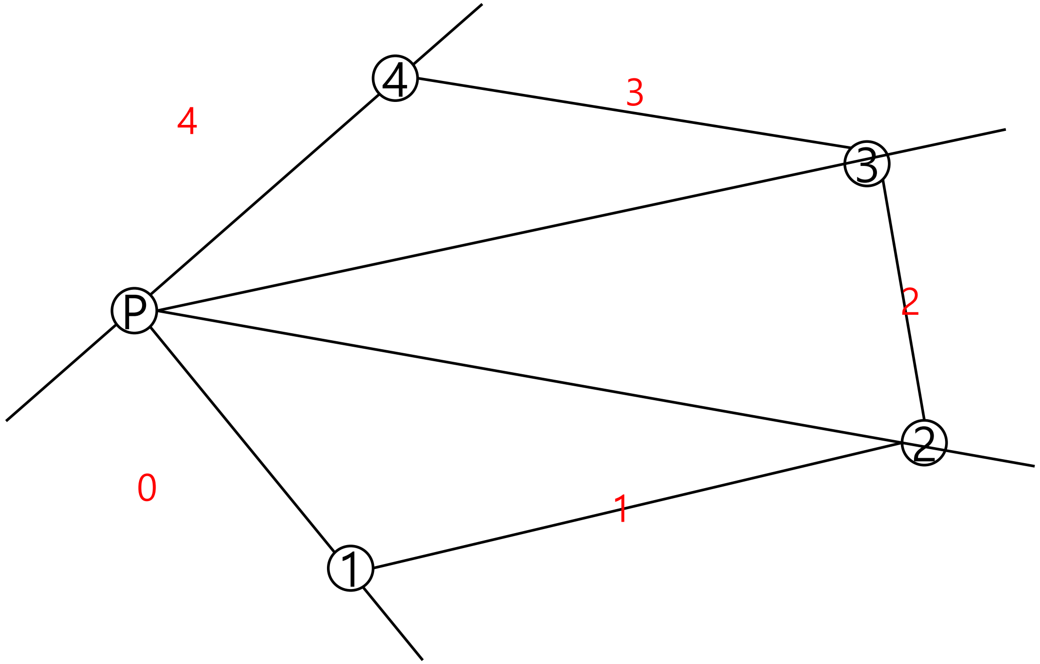
# 볼록 다각형 내부 판별

- 반직선 그려서 교점 세는 방법은 O(N)
  - 볼록 다각형이면 교점 최대 2개 밖에 없는데 굳이 이렇게 해야 할까?



# 볼록 다각형 내부 판별

- 반직선 그려서 교점 세는 방법은 O(N)
  - 볼록 다각형이면 교점 최대 2개 밖에 없는데 굳이 이렇게 해야 할까?
  - 볼록 N각형은 N-2개의 삼각형으로 나타낼 수 있다.
  - 어떤 점이 삼각형 내부에 있는지는 O(1)에 판별 가능 (CCW 3번)
  - 몇 번 영역에 속하는지 찾는 건 이분 탐색을 이용하면 됨



# 볼록 다각형 내부 판별

```
bool Check(const vector<Point> &v, const Point &pt){
    if(CCW(v[0], v[1], pt) < 0) return false;
    int l = 1, r = v.size() - 1;
    while(l < r){
        int m = l + r + 1 >> 1;
        if(CCW(v[0], v[m], pt) >= 0) l = m;
        else r = m - 1;
    }
    if(l == v.size() - 1)
        return CCW(v[0], v.back(), pt) == 0 && v[0] <= pt && pt <= v.back();
    return CCW(v[0], v[l], pt) >= 0
        && CCW(v[l], v[l+1], pt) >= 0
        && CCW(v[l+1], v[0], pt) >= 0;
}
```

# 질문

가장 먼 두 점

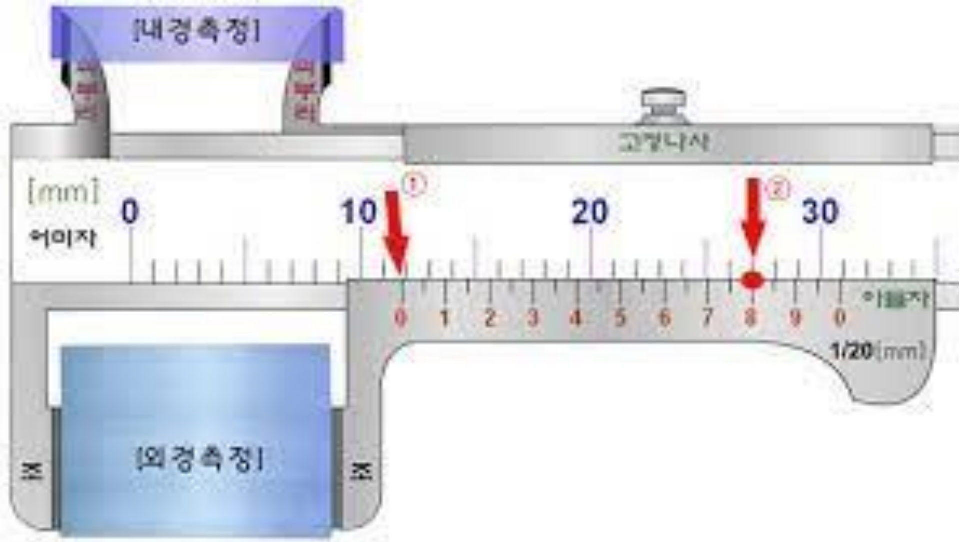
# 가장 먼 두 점

- 가장 먼 두 점은 항상 볼록 껍질의 꼭짓점이다.
  - 일단 O(N log N)에 볼록 껍질을 구하고 시작하자.

# 가장 먼 두 점

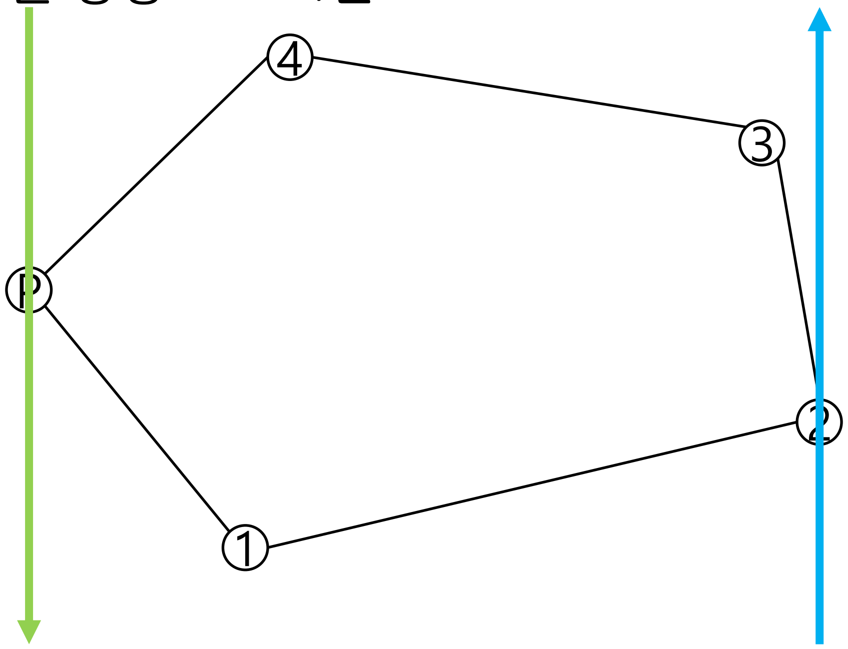
- 가장 먼 두 점은 항상 볼록 껍질의 꼭짓점이다.
  - 일단 O(N log N)에 볼록 껍질을 구하고 시작하자.

- 버니어 캘리퍼스 : 길이를 정밀하게 측정하는 자의 일종
  - 직업 탐구 기초 제도 과목에 나옴ㅎㅎ;;
  - 아 올해 수능부터 기초 제도 없어졌구나

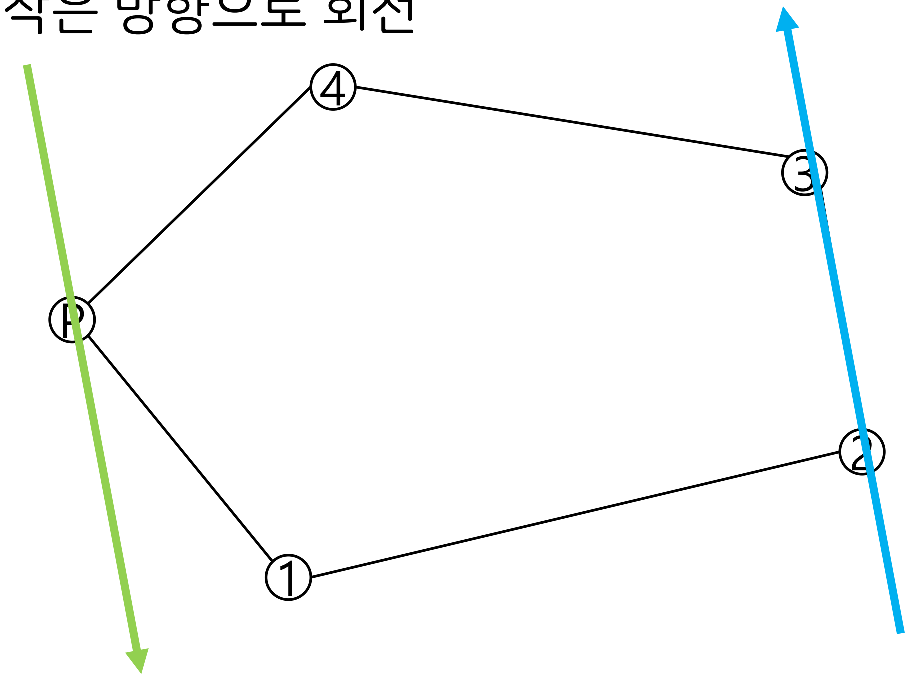
# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



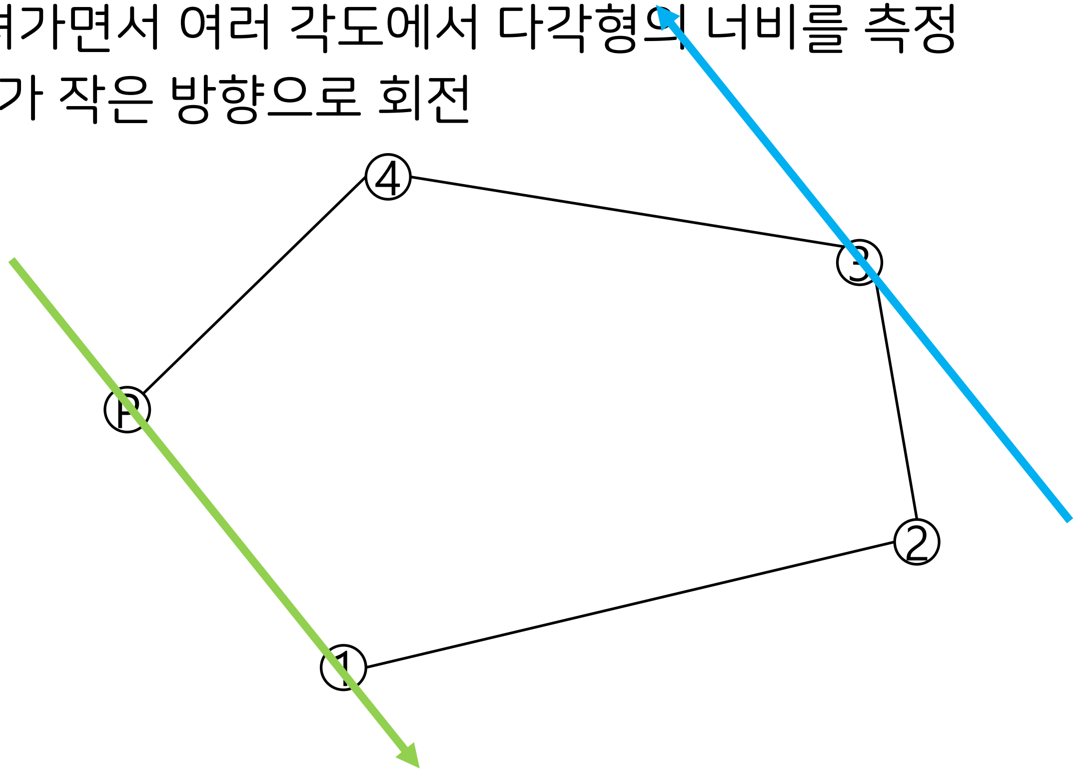
# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



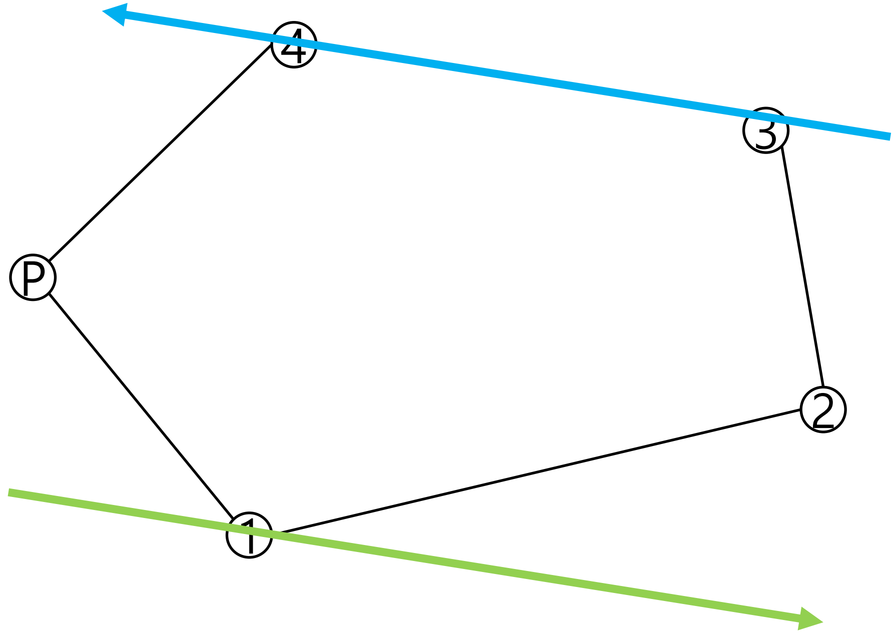
# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



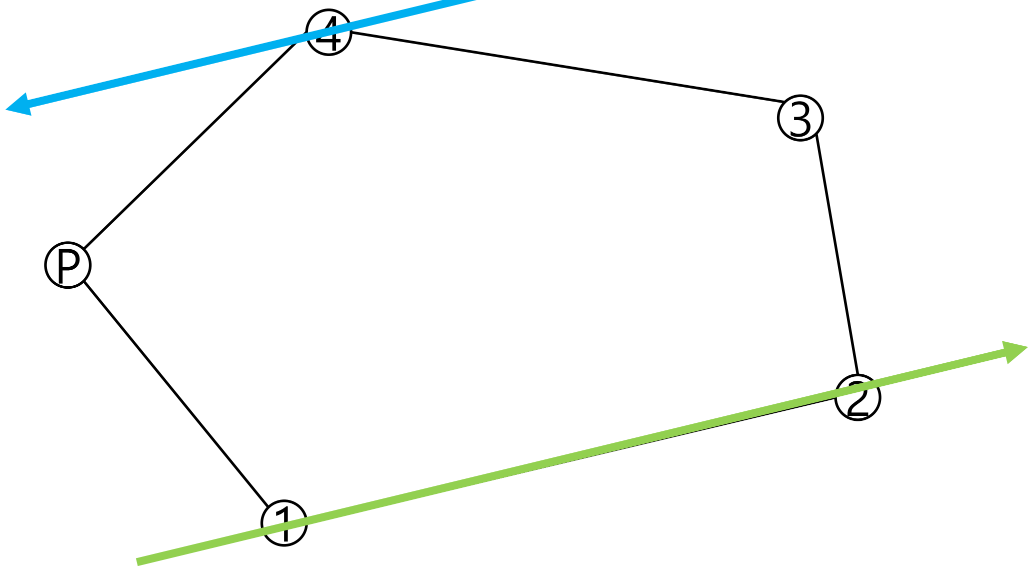
# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



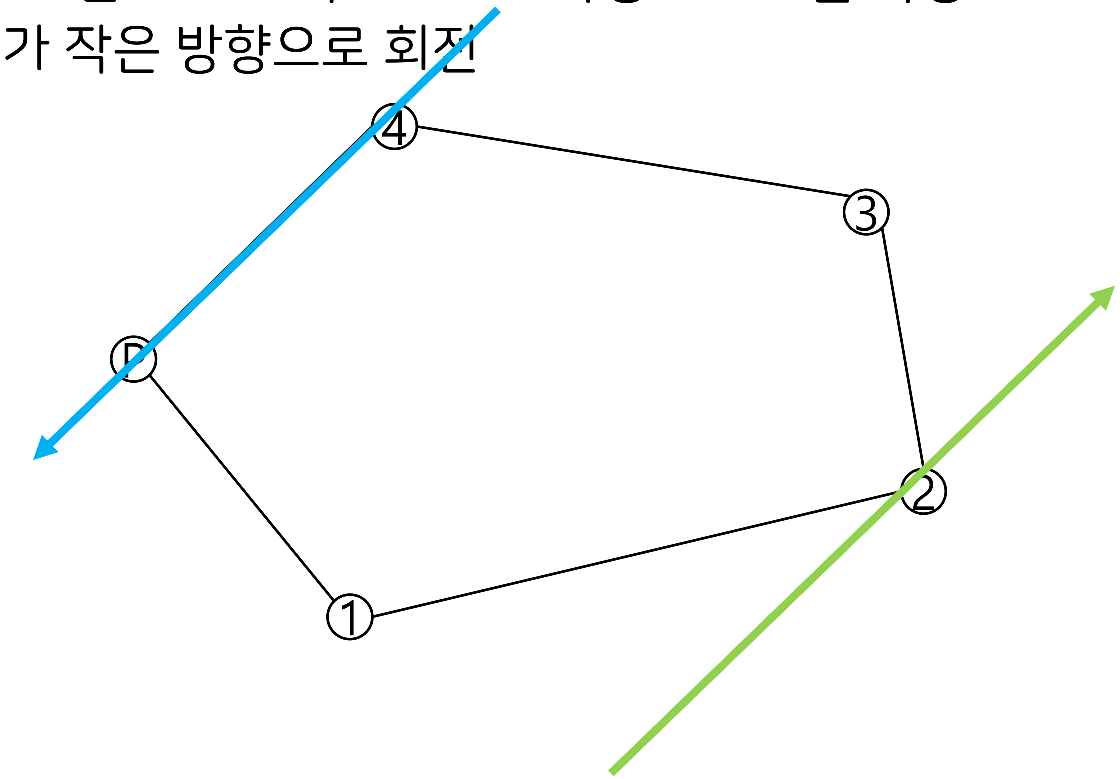
# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



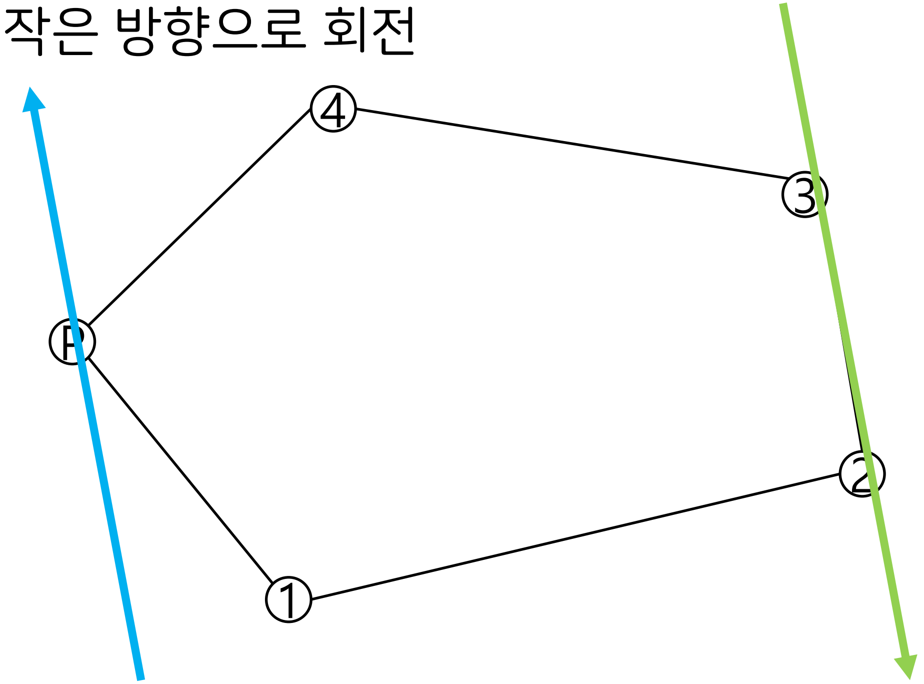
# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전



# 가장 먼 두 점

- Rotating Calipers
  - 캘리퍼스를 돌려가면서 여러 각도에서 다각형의 너비를 측정
  - 두 방향 중 각도가 작은 방향으로 회전

  - 각도를 실수로 직접 계산하는 건 귀찮음
  - 조금만 더 고민해보자.

  - 볼록 껍질을 구할 때 각도 대신 CCW로 비교했던 것처럼
  - 이것도 CCW로 할 수 있지 않을까?

# 가장 먼 두 점

```
Point operator - (const Point &p1, const Point &p2){
    return {p2.x - p1.x, p2.y - p1.y};
}

pair<Point, Point> RotatingCalipers(const vector<Point> &v){
    int n = v.size();
    ll mx = 0; Point a, b;

    for(int i=0, j=0; i<n; i++){
        while(j + 1 < n && CCW(Point(0,0), v[i+1] - v[i], v[j+1] - v[j]) >= 0){
            ll now = Dist(v[i], v[j]);
            if(now > mx) mx = now, a = v[i], b = v[j];
            j++;
        }
        ll now = Dist(v[i], v[j]);
        if(now > mx) mx = now, a = v[i], b = v[j];
    }
    return {a, b};
}
```

# 질문

# 가장 가까운 두 점

# 가장 가까운 두 점

- 생각해보니 1학기에 분할 정복 설명하면서 이미 했음
- 기억 안 나면 깃허브 들어가서 보자.
  - https://github.com/justiceHui/Sunrin-SHARC/blob/master/2021-1st/slide/04.pdf

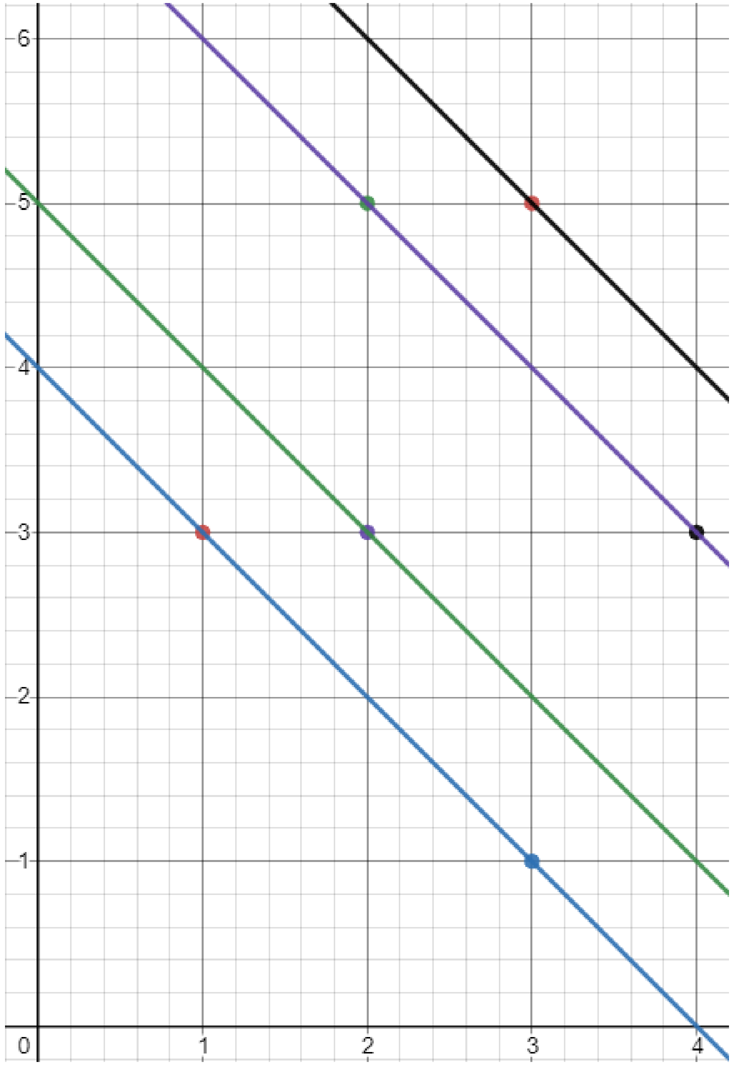
# 볼록 다각형의 접선을 이용한 최적화

# 볼록 다각형의 접선

- N개의 점 (x_i, y_i)가 주어진다.
- 실수 a가 주어지면 a*x_i + y_i의 최댓값을 구하자.
  - http://www.jungol.co.kr/bbs/board.php?bo_table=pbank&wr_id=3019

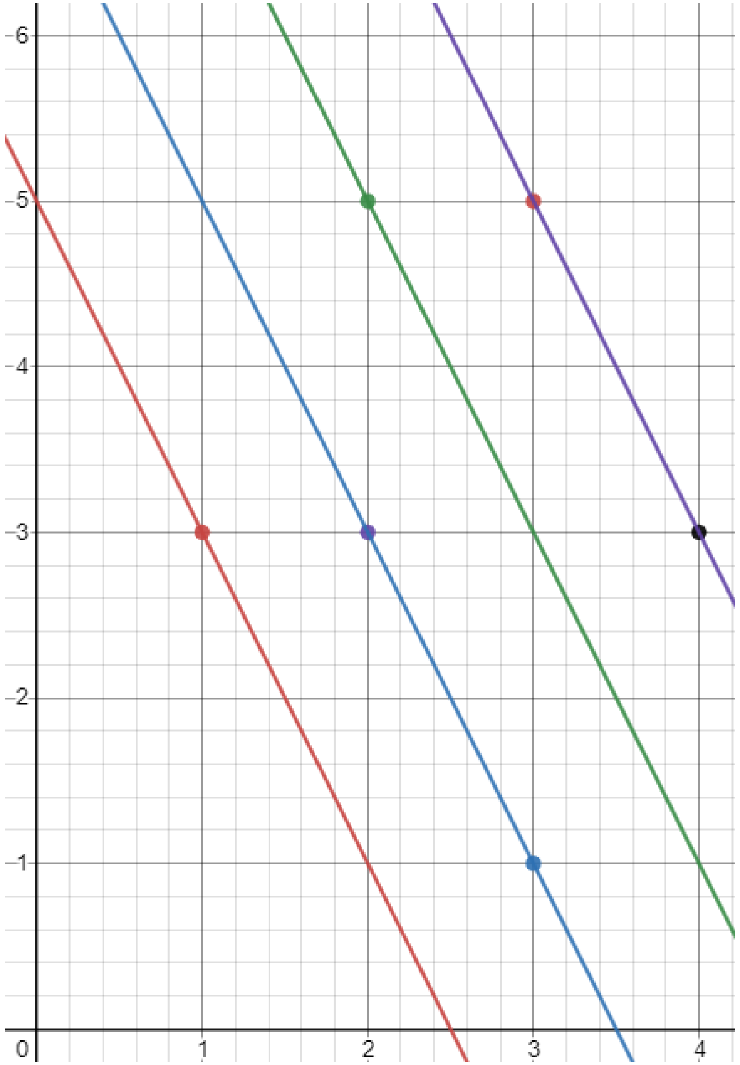
# 볼록 다각형의 접선

- N개의 점 (x_i, y_i)가 주어진다.
- 실수 a가 주어지면 a*x_i + y_i의 최댓값을 구하자.
  - a = 1이면 y_i – x_i가 같은 점들끼리 값이 동일함



# 볼록 다각형의 접선

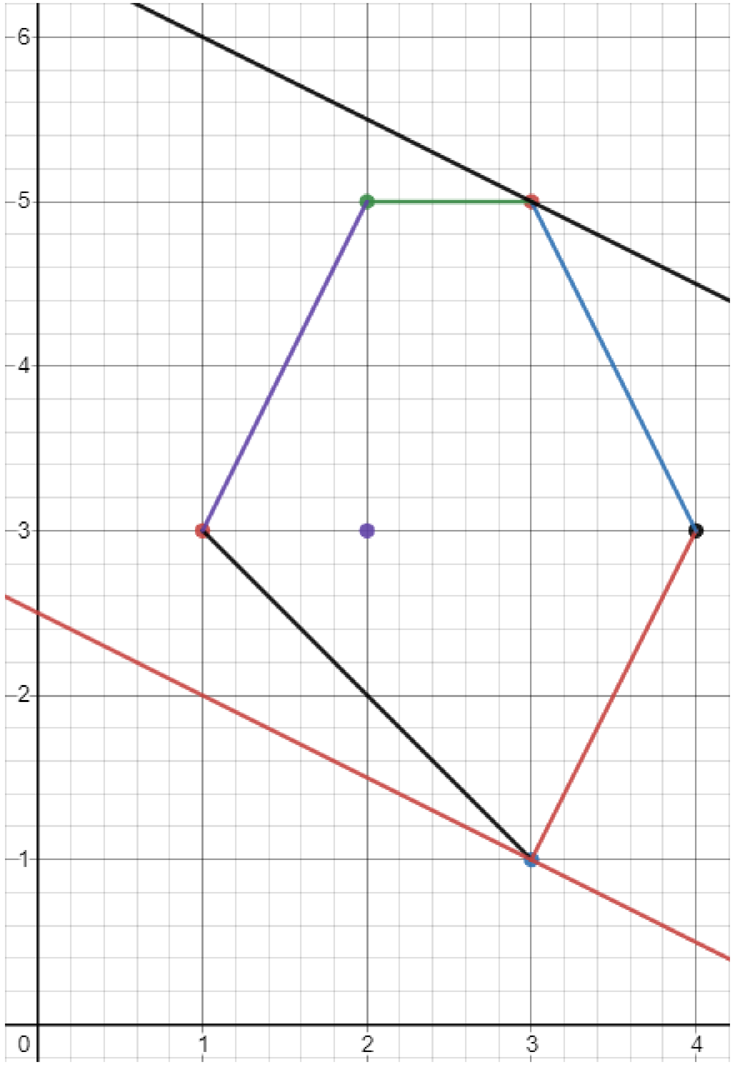
- N개의 점 (x_i, y_i)가 주어진다.
- 실수 a가 주어지면 a*x_i + y_i의 최댓값을 구하자.
  - a = 1이면 y_i – x_i가 같은 점들끼리 값이 동일함
  - a = 2이면 y_i – x_i*2가 같은 점들끼리 값이 동일함



# 볼록 다각형의 접선

- N개의 점 (x_i, y_i)가 주어진다.
- 실수 a가 주어지면 a*x_i + y_i의 최댓값을 구하자.
  - a = 1이면 y_i - x_i가 같은 점들끼리 값이 동일함
  - a = 2이면 y_i - x_i*2가 같은 점들끼리 값이 동일함

  - 볼록 껍질 구하고 기울기가 -a인 접선의 접점
    - 최솟값 : 아래에서 접하는 접선
    - 최댓값 : 위에서 접하는 접선
    - 윗 껍질 / 아랫 껍질 나누고 이분 탐색하면 됨



# 볼록 다각형의 접선

- http://boj.kr/fcf0c311792a41d1961f2a85f141e881

# 볼록 다각형의 접선

- N개의 점 (x_i, y_i)와 실수 a가 주어지면 a*x_i + y_i의 최댓값 구하기

- N개의 일차 함수 f_i(x) = a_i*x + b_i와 실수 x가 주어지면 f_i(x)의 최댓값 구하기

- y = ax + b
  - 일차 함수로 표현
  - 기울기 a = (y2 – y1) / (x1 – x2)
  - 절편 b = y1 – mx1
  - 장점 : 교점을 구하기 쉬움, 점과 동일하게 취급할 수 있음
  - 단점 : 기울기가 무한대일 때 예외 발생, 선분을 표현하기 힘듬

  - ax + by + c = 0으로 표현하는 경우도 존재
  - 기울기 무한대를 표현할 수 있지만 수식이 더러워짐

# 질문

# 볼록 다각형의 접선을 이용한 최적화

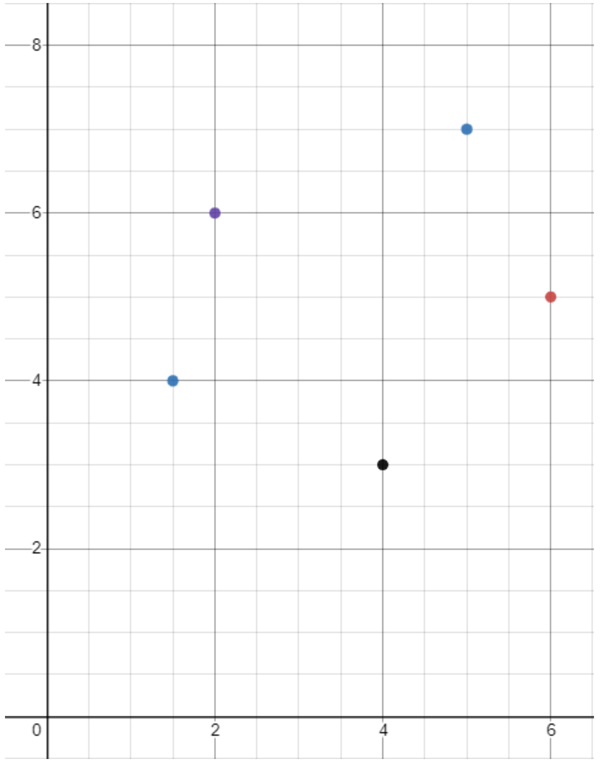
- N개의 점 (x_i, y_i)와 실수 a, b가 주어지면 a*x_i + b*y_i를 최대화/최소화
  - b * (a/b*x_i + y_i)이므로 기울기가 –a/b인 접선
  - 이걸 쓸 일이 있을까?

# 볼록 다각형의 접선을 이용한 최적화

- N개의 점 (x_i, y_i)와 실수 a, b가 주어지면 a*x_i + b*y_i를 최대화/최소화
  - b * (a/b*x_i + y_i)이므로 기울기가 –a/b인 접선
  - 이걸 쓸 일이 있을까?

  - 각 원소는 두 가지 종류의 가중치 A_i, B_i를 갖고 있고
  - 이들 중 몇 개를 선택해서 (sum A_i) * (sum B_i)의 합을 최소화하는 문제

  - 모든 경우에 대해, 좌표 평면에 (sum A_i, sum B_i) 점을 찍어보자

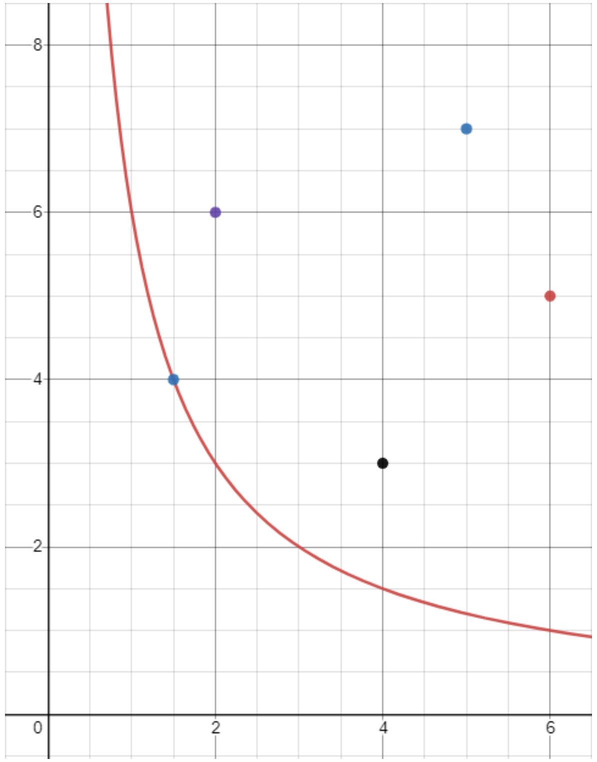


# 볼록 다각형의 접선을 이용한 최적화

- N개의 점 (x_i, y_i)와 실수 a, b가 주어지면 a*x_i + b*y_i를 최대화/최소화
  - b * (a/b*x_i + y_i)이므로 기울기가 –a/b인 접선
  - 이걸 쓸 일이 있을까?

  - 각 원소는 두 가지 종류의 가중치 A_i, B_i를 갖고 있고
  - 이들 중 몇 개를 선택해서 (sum A_i) * (sum B_i)의 합을 최소화하는 문제

  - 모든 경우에 대해, 좌표 평면에 (sum A_i, sum B_i) 점을 찍어보자
  - 최솟값이 c라면 다른 모든 점들은 xy = c 곡선의 위쪽에 존재함

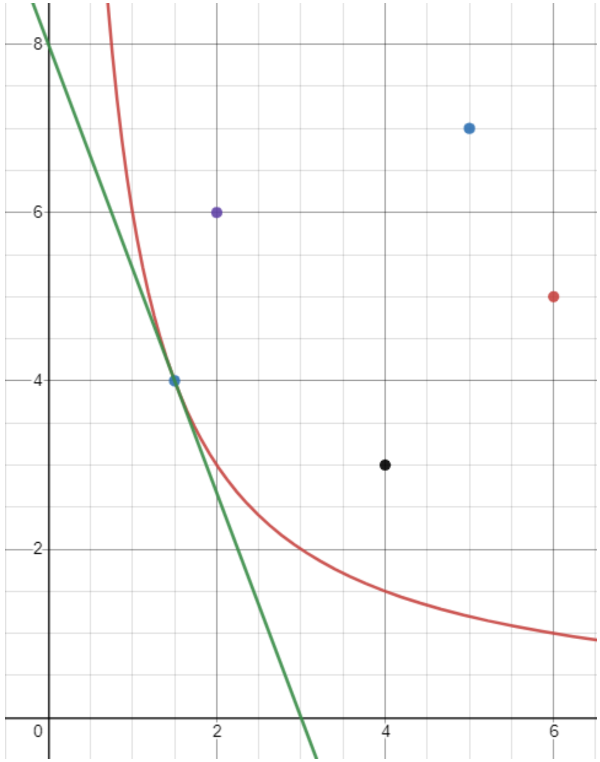


# 볼록 다각형의 접선을 이용한 최적화

- N개의 점 (x_i, y_i)와 실수 a, b가 주어지면 a*x_i + b*y_i를 최대화/최소화
  - b * (a/b*x_i + y_i)이므로 기울기가 –a/b인 접선
  - 이걸 쓸 일이 있을까?

  - 각 원소는 두 가지 종류의 가중치 A_i, B_i를 갖고 있고
  - 이들 중 몇 개를 선택해서 (sum A_i) * (sum B_i)의 합을 최소화하는 문제

  - 모든 경우에 대해, 좌표 평면에 (sum A_i, sum B_i) 점을 찍어보자
  - 최솟값이 c라면 다른 모든 점들은 xy = c 곡선의 위쪽에 존재함
  - 답이 되는 점을 A(x, y)라고 하면
  - ax + by = c가 되도록 하는 a, b가 존재 (기울기가 –b/a인 접선)
  - 모든 기울기에 대해 접점을 구한 뒤, 그 중 최솟값을 취하면 됨



# 볼록 다각형의 접선을 이용한 최적화

- 고려해야 하는 기울기 : 볼록 껍질에서 변의 기울기
  - 고려해야 하는 점의 개수가 너무 많아서 볼록 껍질을 직접 구할 수 없음
  - 볼록 껍질 위의 점의 개수 : min{ (좌표 범위)^(2/3), N }
  - 기울기가 주어졌을 때, 그 기울기에 대한 접점을 T(N) 시간에 구할 수 있다면 X^(2/3)T(N)
    - ex) X = N이고 T(N) = N log N이면 전체 시간 복잡도는 O(N^(5/3) log N)
    - 대충 N^2이 될 것 같은 입력 제한이면 의심해보자

# 질문

# 볼록 다각형의 접선을 이용한 최적화

- BOJ 5257 timeismoney
  - MST 비슷한 것을 만드는 문제
  - 각 간선은 X_i, Y_i라는 두 가지 종류의 가중치가 있음
  - 스패닝 트리의 가중치는 (sum X_i) * (sum Y_i)로 정의함
  - 가중치가 최소인 스패닝 트리를 구하는 문제

  - 1 ≤ N ≤ 200
  - 1 ≤ M ≤ 10’000
  - 1 ≤ X_i, Y_i ≤ 255

  - 좌표 범위는 200 * 255 = 51000
  - 51000^(2/3) ≤ 1’400

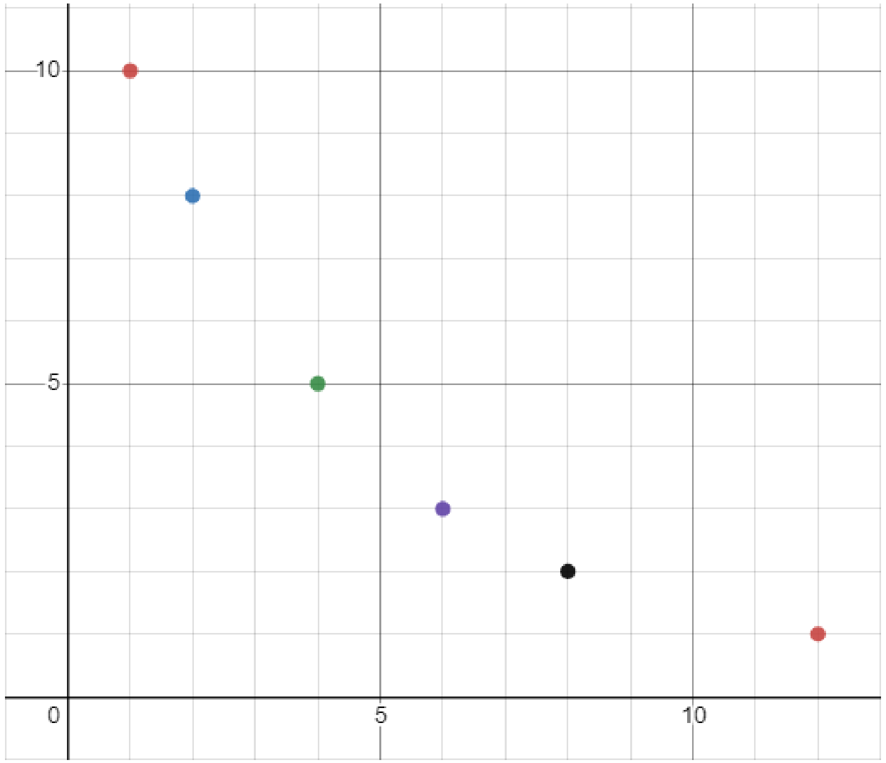
# 볼록 다각형의 접선을 이용한 최적화

- 기울기 –a/b가 주어지면 접점을 구하는 방법
  - a*(sum X_i) + b*(sum Y_i)를 최소화 해야 하므로
  - 간선을 aX_i + bY_i순으로 정렬하고 크루스칼
  - O(M log M)이므로 1400 * M log M에 문제를 풀 수 있다!

- 필요한 기울기를 어떻게 빠르게 구하지?

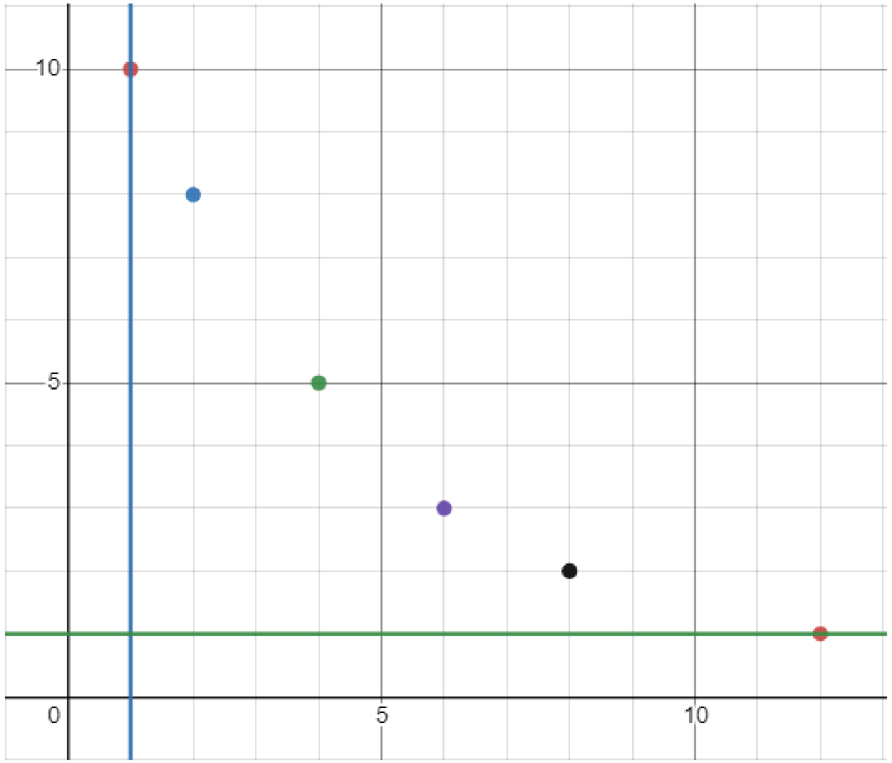
# 볼록 다각형의 접선을 이용한 최적화

- 가장 왼쪽에 있는 점 / 가장 아래에 있는 점은 쉽게 구할 수 있음



# 볼록 다각형의 접선을 이용한 최적화

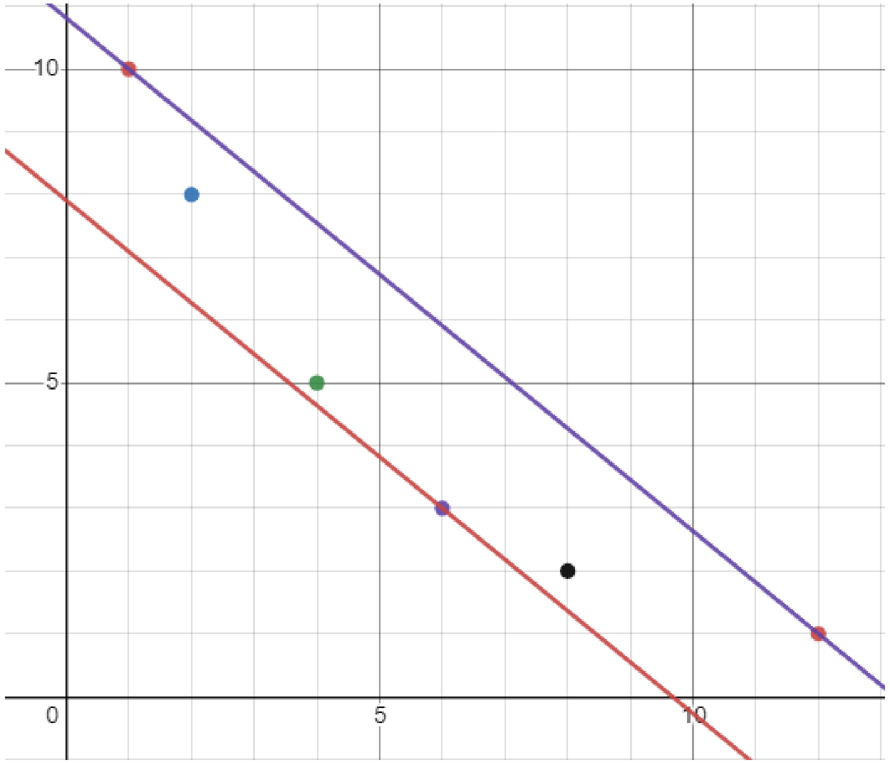
- 가장 왼쪽에 있는 점 / 가장 아래에 있는 점은 쉽게 구할 수 있음
  - 기울기가 1/0인 접선과 0/1인 접선



# 볼록 다각형의 접선을 이용한 최적화

- 가장 왼쪽에 있는 점 / 가장 아래에 있는 점은 쉽게 구할 수 있음
  - 기울기가 1/0인 접선과 0/1인 접선

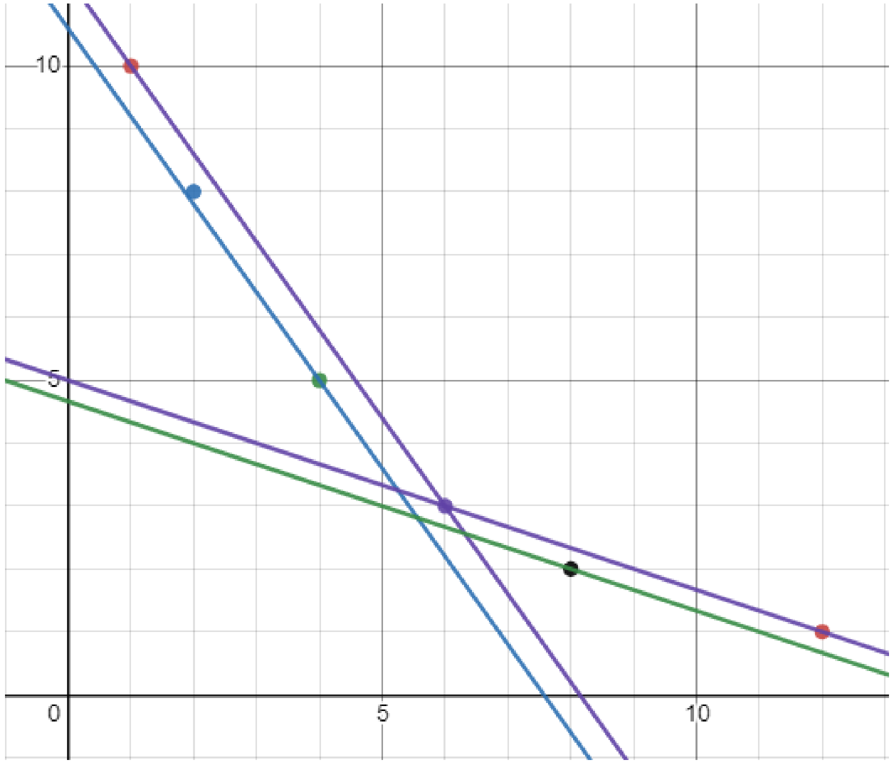
- 두 점을 잇는 직선의 기울기의 접점을 구하자



# 볼록 다각형의 접선을 이용한 최적화

- 가장 왼쪽에 있는 점 / 가장 아래에 있는 점은 쉽게 구할 수 있음
  - 기울기가 1/0인 접선과 0/1인 접선

- 두 점을 잇는 직선의 기울기의 접점을 구하자
  - 가장 왼쪽에 있는 점과 지금 찾은 점을 잇는 직선
  - 가장 아래에 있는 점과 지금 찾은 점을 잇는 직선
  - …
  - 분할 정복
    - 볼록 껍질 위의 점 개수 만큼만 호출됨



# 볼록 다각형의 접선을 이용한 최적화

```cpp
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;
using ll = long long;
using Point = pair<ll, ll>;

ll CCW(const Point &p1, const Point &p2, const Point &p3){
    return (p2.x - p1.x) * (p3.y - p2.y) - (p3.x - p2.x) * (p2.y - p1.y);
}

struct Edge{ int u, v, x, y; };
struct UnionFind{
    int P[222];
    UnionFind(){ clear(); }
    void clear(){ iota(P, P+222, 0); }
    int find(int v){ return v == P[v] ? v : P[v] = find(P[v]); }
    bool merge(int u, int v){
        u = find(u); v = find(v);
        if(u == v) return false;
        P[u] = v; return true;
    }
};

int N, M;
Edge E[10101];
UnionFind UF;
vector<pair<int,int>> MST;
ll OptX = 1e9, OptY = 1e9;
```

```cpp
Point Optimize(ll dy, ll dx){
    UF.clear();
    sort(E+1, E+M+1, [&](const Edge &a, const Edge &b){
        return dy*a.x + dx*a.y < dy*b.x + dx*b.y;
    });
    vector<pair<int,int>> now;
    ll sx = 0, sy = 0;
    for(int i=1; i<=M; i++){
        if(UF.merge(E[i].u, E[i].v)){
            sx += E[i].x; sy += E[i].y;
            now.emplace_back(E[i].u, E[i].v);
        }
    }
    if(sx*sy < OptX*OptY) OptX = sx, OptY = sy, MST = now;
    return { sx, sy };
}

void Solve(Point le, Point dw){
    Point pt = Optimize(le.y - dw.y, dw.x - le.x);
    if(CCW(le, pt, dw) > 0) Solve(le, pt), Solve(pt, dw);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=M; i++) cin >> E[i].u >> E[i].v >> E[i].x >> E[i].y;

    auto le = Optimize(1, 0), dw = Optimize(0, 1);
    Solve(le, dw);
    cout << OptX << " " << OptY << "\n\n";
    for(auto i : MST) cout << i.x << " " << i.y << "\n";
}
```

# 질문

# 더 공부할 거리

- 만약 이게 재밌다면…
  - K-D Tree
  - Rotate Sweep Line (A.K.A. Bulldozer Trick)
  - Half Plane Intersection
  - Dual Graph